Методические указания к лабораторным работам по курсу

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Содержание

[Лабораторная работа № 6](#_Лабораторная_работа_№)

[Лабораторная работа № 7](#_Лабораторная_работа_№_1)

[Лабораторная работа № 8](#_Лабораторная_работа_№_2)

[Лабораторная работа № 9](#_Лабораторная_работа_№_3)

[Лабораторная работа № 10](#_Лабораторная_работа_№_4)

### Лабораторная работа № 6

**Исследование файловой системы NTFS.**

***Цель работы:*** изучить структуру файловой системы NTFS и получить навыки программного доступа к элементам этой файловой системы с помощью функций WIN API.

**Теоретические сведения**

В NTFS все пространство разделено на кластера. Кластером называется группа смежных секторов, количество которых равно степени 2 (то есть 1, 2, 4, 8, 16). Каждому кластеру назначается адрес, начиная с 0. Чтобы преобразовать адрес кластера в адрес сектора, достаточно умножить его на количество секторов в кластере:

СЕКТОР =КЛАСТЕР \*секторов\_в\_кластере.

Для обозначения кластеров используются два типа номеров: LCN и VCN. При помощи первого типа, LCN (Logical Cluster Number – логический номер кластера), нумеруются все кластеры на диске, от первого до последнего. Номера VCN (Virtual Cluster Number – виртуальный номер кластера) обозначают порядковый номер кластера внутри файла.

«Сердцем» NTFS является главная файловая таблица MFT (Master File Table), содержащая информацию обо всех файлах и каталогах. Каждый файл и каталог представлен как минимум одной записью таблицы, причем записи сами по себе очень просты. Их размер составляет 1 Кбайт, но только первые 42 байта имеют определенное предназначение. В остальных байтах хранятся атрибуты — небольшие структуры данных, выполняющие строго специализированную функцию. Например, один атрибут используется для хранения имени файла, а другой — для хранения его содержимого. Каждой записи на основании ее местонахождения в таблице присваивается адрес, начиная с 0. MFT, как и все структуры NTFS, представляет собой таблицу. Ситуация несколько усложняется тем, что MFT содержит запись для представления себя самой. Первая запись таблицы называется $MFT и задает местонахождение MFT на диске. Фактически это единственное место, в котором указывается местонахождение MFT; следовательно, эту запись необходимо обработать для определения структуры и размера MFT. Начальный адрес MFT задается в загрузочном секторе, который всегда располагается в первом секторе файловой системы.

***Файловые записи (дескрипторы)***

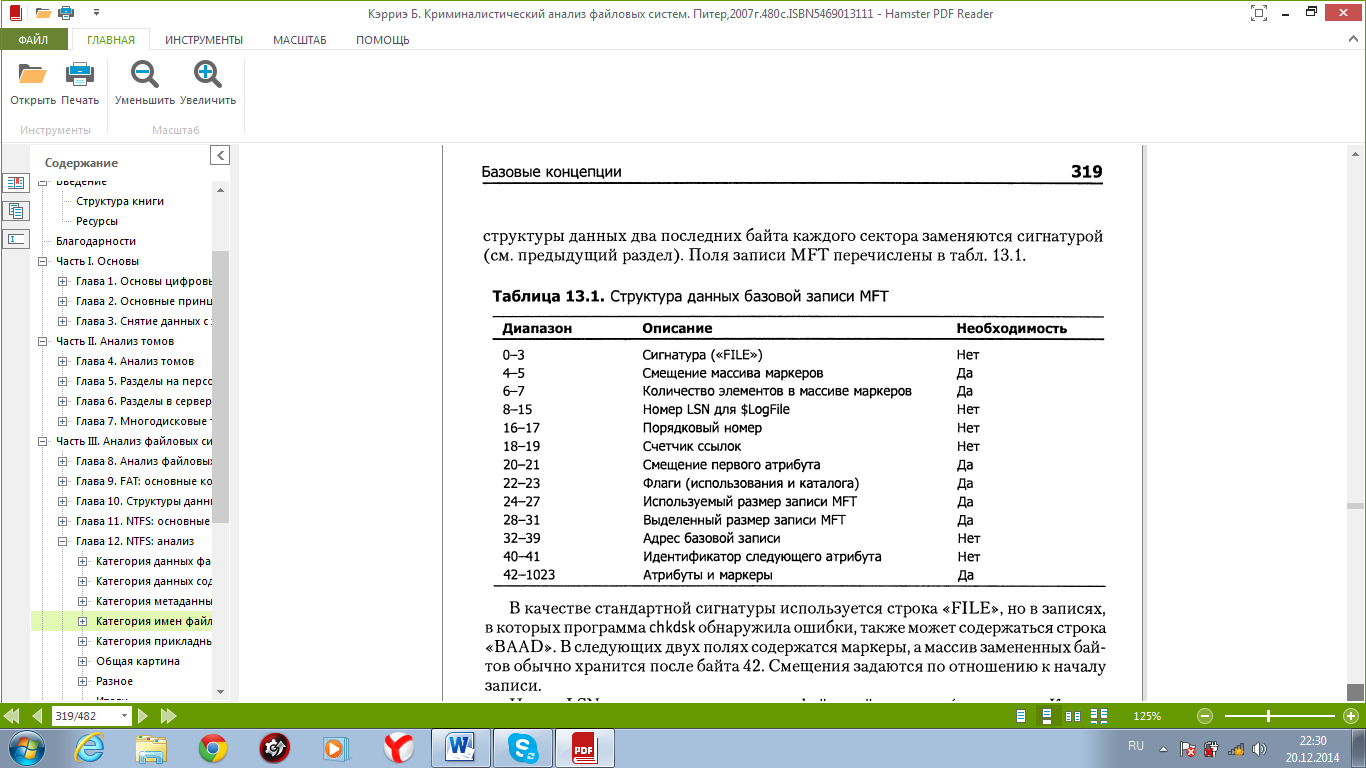
Файловая запись состоит из заголовка (Header) и набора атрибутов (Attribute). В заголовке содержится служебная информация о файловой записи, например, её тип и размер. Все данные, относящиеся непосредственно к файлу, хранятся в виде атрибутов. Названия атрибутов, так же как и системных файлов, начинаются с «$». Например, отдельными атрибутами являются имя файла ($FILE\_NAME), информация о его свойствах ($STANDARD\_INFORMATION), данные файла ($DATA). На диске файловая запись всегда расположена в начале сектора. Чаще всего записи MFT занимают 2 или 4 сектора, хотя встречаются записи и другого размера.

В NTFS файл может целиком размещаться в записи таблицы MFT, если это позволяет сделать его размер. В том же случае, когда размер файла больше размера записи MFT, в запись помещаются только некоторые атрибуты файла, а остальная часть файла размещается в отдельном отрезке тома (или нескольких отрезках). Часть файла, размещаемая в записи MFT, называется резидентной частью (резидентным атрибутом), а остальные части — нерезидентными атрибутами. Адресная информация об отрезках, содержащих нерезидентные части файла, размещается в атрибутах резидентной части.

Некоторые системные файлы являются полностью резидентными, а некоторые имеют и нерезидентные части, которые располагаются после MFT. Если значение атрибута хранится непосредственно в MFT, время, необхо­димое NTFS для доступа к нему, значительно сокращается.



Структура заголовка файловой записи показана в табл.13.1:



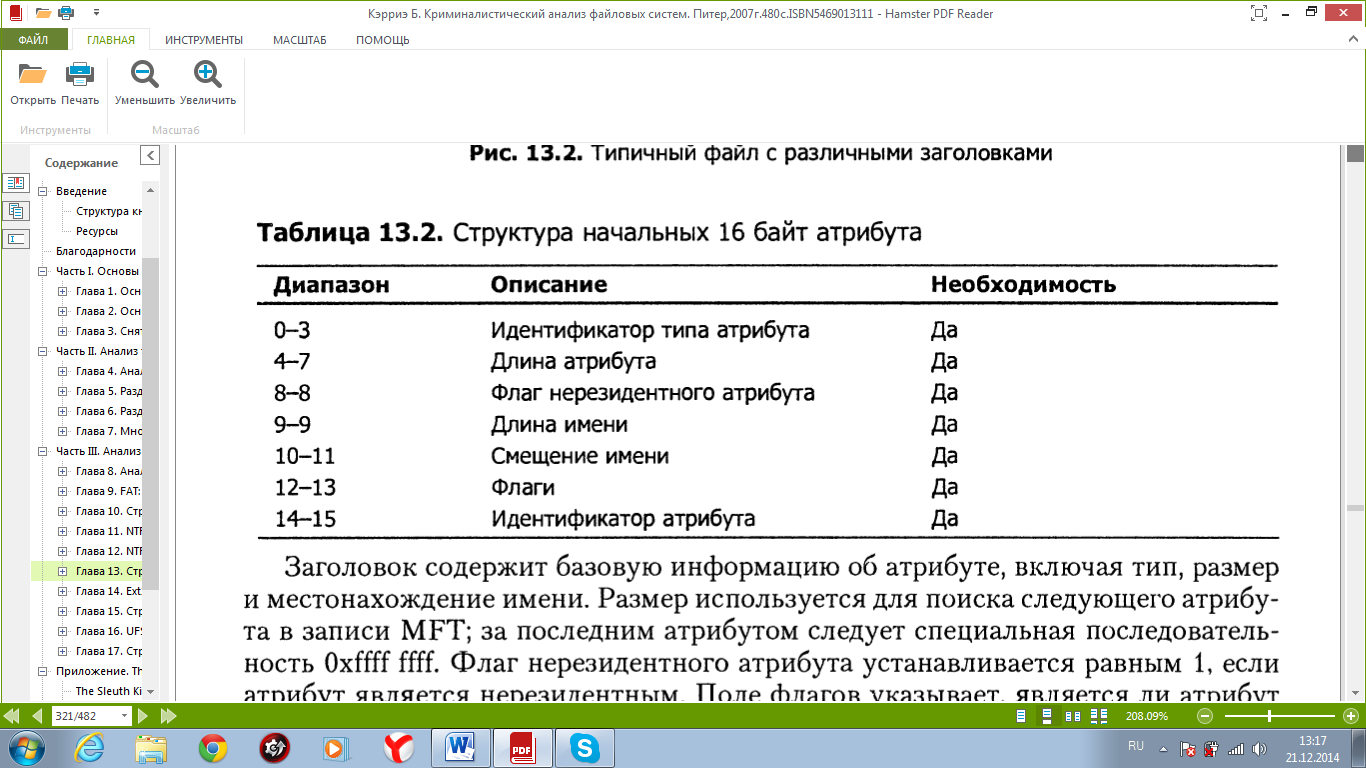
В первом поле каждой записи MFT хранится сигнатура; у стандартных записей это ASCII-строка «FILE». Если в записи обнаружена ошибка, в качестве сигнатуры может использоваться строка «BAAD». В следующих двух полях содержатся маркеры, а массив замененных байтов обычно хранится после байта 42. Смещения задаются по отношению к началу записи.

Первый атрибут файла находится по смещению, заданному относительно начала файла. За первым атрибутом следуют остальные; чтобы найти их, следует сместиться вперед на величину, указанную в поле размера в заголовке атрибута. За последним атрибутом находится признак конца файла Oxffffffff. Если файлу требуется более одной записи MFT, то в дополнительные записи включается базовый адрес основной записи.

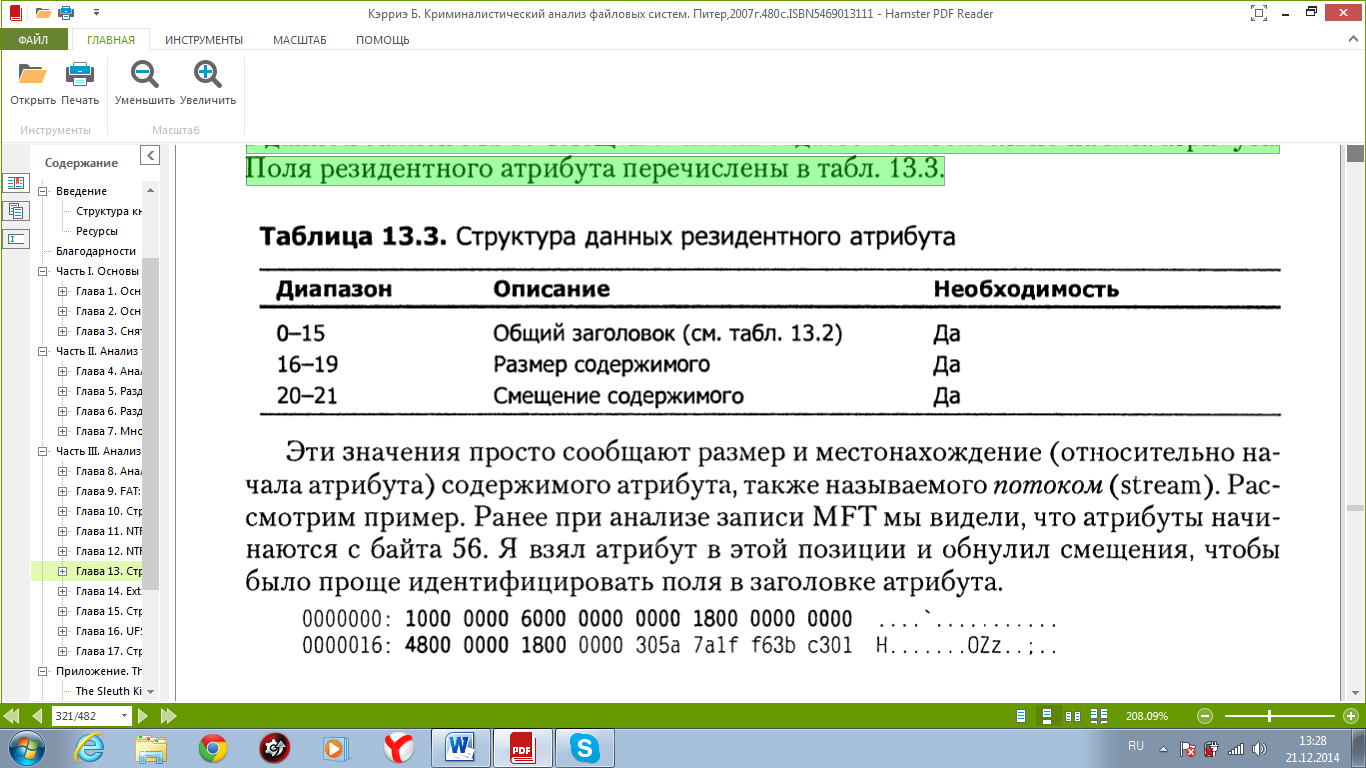
Поле флагов указывает, используется ли запись и представляет ли она каталог. Бит 0x01 устанавливается в том случае, если запись используется, а бит 0x02 —если запись представляет каталог. Если атрибуты не помещаются в одной записи, для файла создаются несколько записей MFT. В этом случае первая запись называется базовой записью MFT, а ее адрес сохраняется в одном из фиксированных полей всех остальных записей.

***Заголовки атрибутов***

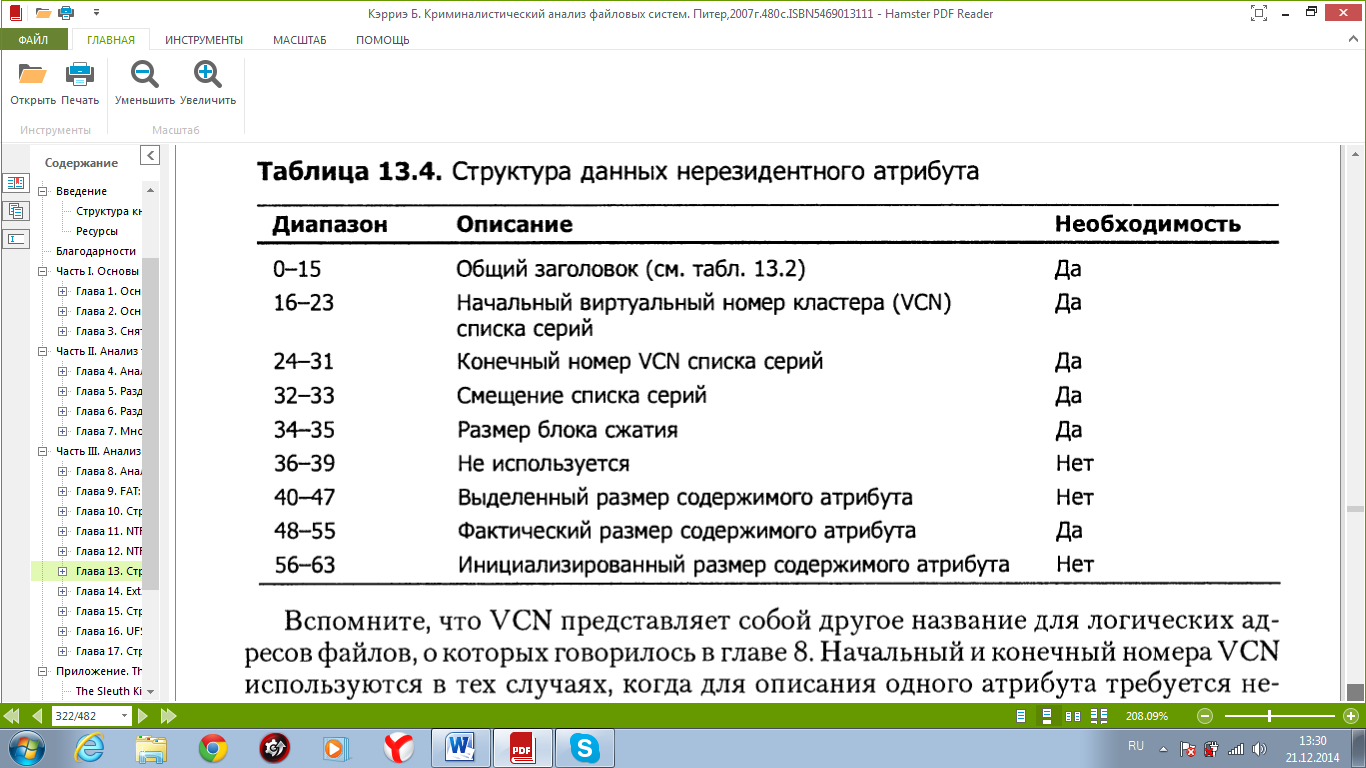
У резидентных и нерезидентных атрибутов структуры данных слегка различаются, потому что в нерезидентных атрибутах требуется дополнительно хранить информацию о сериях. Первые 16 байт атрибутов обоих типов совпадают. Содержащиеся в них поля перечислены в табл. 13.2. Заголовок содержит базовую информацию об атрибуте, включая тип, размер и местонахождение имени. Размер используется для поиска следующего атрибута в записи MFT; за последним атрибутом следует специальная последовательность Oxffff ffff. Флаг нерезидентного атрибута устанавливается равным 1, если атрибут является нерезидентным. Поле флагов указывает, является ли атрибут сжатым (0x00001), зашифрованным (0x4000) или разреженным (0x8000). Идентификатор атрибута представляет собой число, уникальное для данного атрибута в данной записи MFT. Если запись содержит несколько однотипных атрибутов, они различаются по значению идентификатора.



Смещение имени задается относительно начала атрибута. Некоторым атрибутам назначается имя, хранящееся в кодировке Unicode UTF-16 в заголовке атрибута. Имена используются в основном с атрибутами данных для идентификации второго или третьего потока данных в файле. Поля резидентного атрибута перечислены в табл. 13.3.

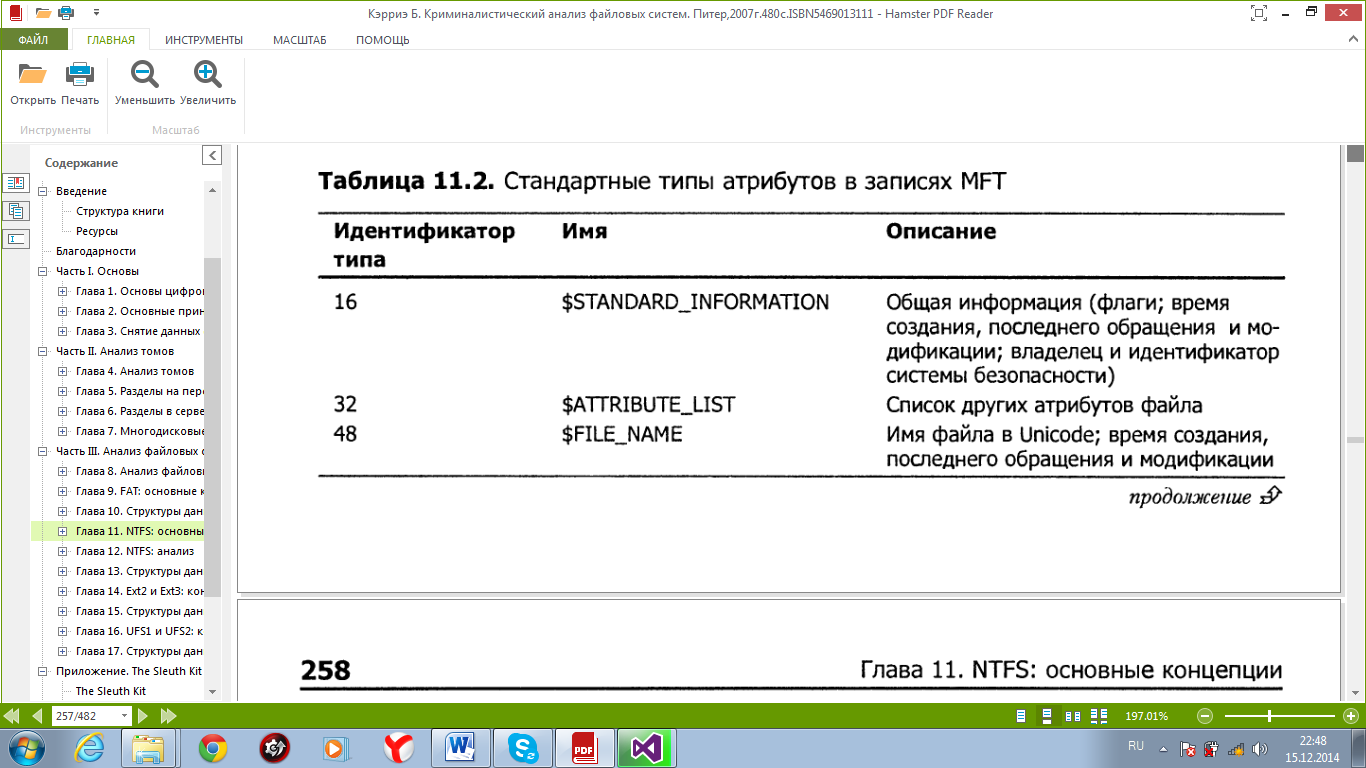


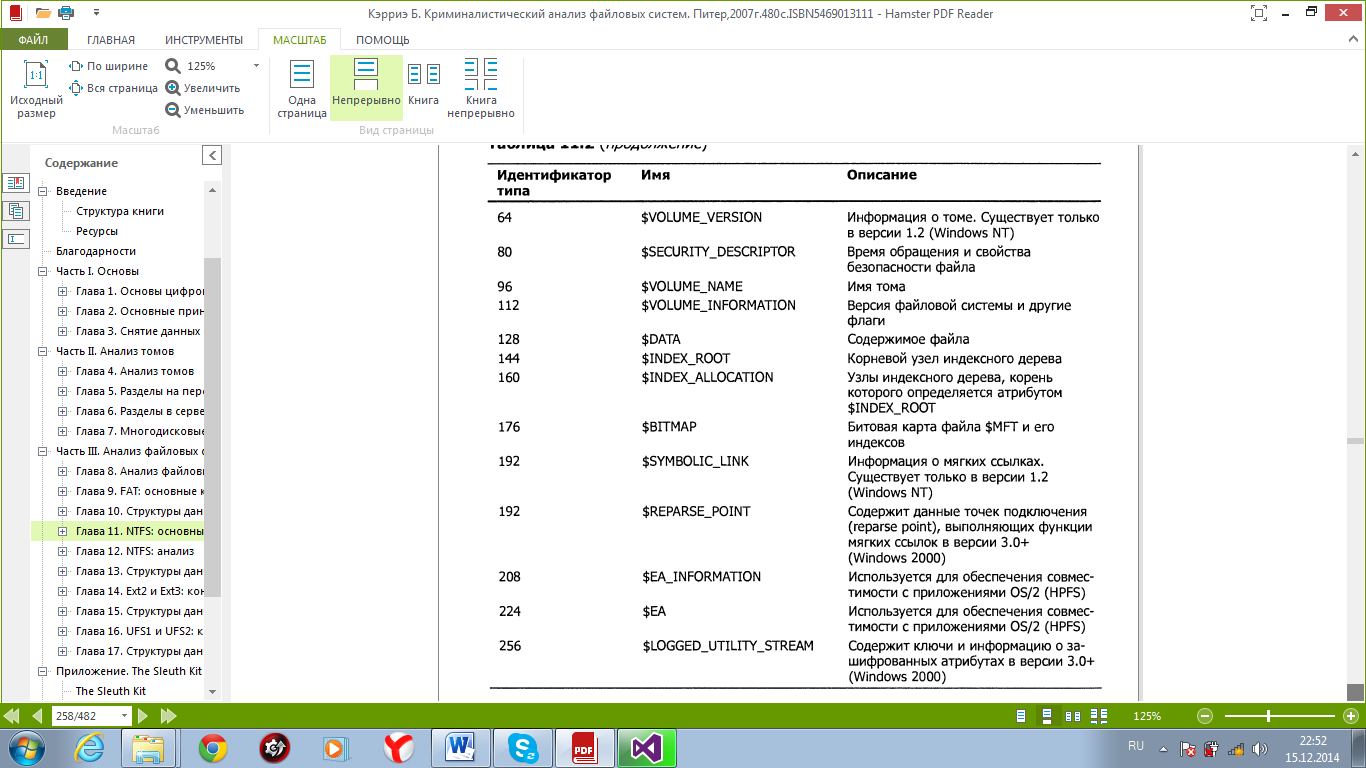
Поля структуры данных нерезидентных атрибутов приведены в табл. 13.4.



***Стандартные типы атрибутов***

Каждый тип атрибута представляется неким числом, причем Microsoft сортирует атрибуты в записи по этому числу. Стандартным атрибутам присваивается значение по умолчанию, но это значение можно переопределить при помощи файла метаданных файловой системы $AttrDef. Кроме числового идентификатора каждый тип атрибута обладает именем, которое состоит только из прописных букв и начинается со знака «$». В табл. 11.2 перечислены некоторые стандартные типы атрибутов и соответствующие им идентификаторы. Учтите, что не все типы атрибутов и идентификаторы существуют для каждого файла.





Почти все выделенные записи MFT содержат атрибуты типов $FILE\_NAME и $STANDARD\_INFORMATION. Единственное исключение составляют не-базовые записи MFT, о которых речь пойдет далее. Атрибут $FILE\_NAME содержит имя файла, размер и временные штампы. Атрибут $STANDARD\_INFORMATION содержит временные штампы, информацию о владельцах и безопасности. Последний существует во всех файлах и каталогах, потому что содержащиеся в нем данные необходимы для обеспечения безопасности данных и дисковых квот. В абстрактном понимании этот атрибут не содержит необходимых данных, но его присутствие необходимо для работы механизмов файловой системы прикладного уровня. Оба атрибута всегда являются резидентными.

**Файлы метаданных файловой системы**

Поскольку каждый байт тома выделяется некоторому файлу, административные данные файловой системы также должны храниться в файлах. Microsoft называет такие файлы файлами метаданных. Microsoft резервирует для файлов метаданных файловой системы первые 16 записей MFT. Неиспользуемые зарезервированные записи находятся в выделенном состоянии и содержат только базовую и общую информацию. Все файлы метаданных файловой системы отображаются в корневом каталоге, хотя обычно они скрываются от большинства пользователей. Имена файлов метаданных файловой системы начинаются с символа «$», а первая буква является прописной.

Таблица 11.1. Стандартные файлы метаданных файловой системы NTFS

Запись Имя файла Описание

0 $MFT Запись для самой таблицы MFT

1 $MFTMirr Содержит резервную копию первых записей MFT.

2 $LogFile Содержит журнал транзакций метаданных.

3 $Volume Содержит информацию о томе —метка, идентификатор и версия.

4 $AttrDef Содержит инфор-ю об атрибутах — значения идент-ра, имена, размеры.

5 . Содержит корневой каталог файловой системы.

6 $Bitmap Содержит признак выделения для каждого кластера файловой системы.

7 $Boot Содержит загрузочный сектор и загрузочный код файловой системы.

8 $BadClus Содержит кластеры, содержащие поврежденные секторы.

9 $Secure Содержит инфор-ю системы безопасности и упр-я доступом к файлам

10 $Upcase Содержит все символы Unicode в верхнем регистре

11 $Extend Каталог с файлами необязательных расширений. Microsoft обычно не размещает файлы этого каталога в зарезервированных записях MFT.

***Порядок работы с NTFS***

В NTFS первым шагом должна стать обработка загрузочного сектора в первом секторе файловой системы, являющегося частью файла $Boot. Из загрузочного сектора берутся сведения о начальном адресе MFT и размере каждой записи MFT. На основании этих сведений обрабатывается первая запись MFT, связанная с файлом $MFT. Из этой записи берется информация о местонахождении других записей MFT.

Все записи имеют одинаковый размер, поэтому найти запись с нужным номером несложно. После успешного поиска записи в таблице можно переходить к обработке атрибутов. Обработка записи MFT начинается с заголовка, после чего ищется первый атрибут. Далее мы читаем заголовок атрибута, определяем его тип и соответствующим образом обрабатываем содержимое. Местонахождение второго атрибута определяется по длине первого атрибута. Процедура повторяется до тех пор, пока не будут обработаны все атрибуты. За последним атрибутом следует маркер Oxffffffff.

**Пример упрощенного чтения нефрагментированной MFT**

#include <Windows.h>

#include <stdio.h>

int main()

{

struct{

DWORD TypeID;

UINT Length;

char Resident;

char NameLength;

WORD NameOffset;

WORD Flags;

WORD ID;

DWORD SizeContent;

WORD OffsetContent;

}HeaderAttrib;

struct{

char ParentDir[8];

UINT64 TimeCreate;

UINT64 TimeModif;

UINT64 TimeChange;

UINT64 TimeAcces;

UINT64 Size;

UINT64 RealSize;

UINT Attr;

UINT res;

char NameLength;

char NameSpace;

char FileName[100];

}NameAttr;

char buf[1024];

DWORD dwByteRead;

LARGE\_INTEGER BeginMFTSector;

HANDLE h = CreateFile("\\\\.\\D:", GENERIC\_READ | GENERIC\_WRITE, FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, NULL, OPEN\_EXISTING, 0, NULL);

if (h != INVALID\_HANDLE\_VALUE)

{

ReadFile(h, buf, 512, &dwByteRead, NULL);//read BOOT

UINT64 s = \*((long long \*)(buf + 48));

//UINT64 BeginMFTSector

BeginMFTSector.QuadPart = buf[13] \* s \* 512;

while (1){//cicle MFT

DWORD dwLow = SetFilePointer(h, BeginMFTSector.LowPart, (PLONG)&BeginMFTSector.HighPart, FILE\_BEGIN);

ReadFile(h, buf, 1024, &dwByteRead, NULL);//file record

if (strncmp(buf, "FILE",4)) break;

WORD AttrOffset=\*(WORD \*)(buf + 20);

while (1)//cicle attrib

{

memcpy(&HeaderAttrib, buf + AttrOffset, sizeof(HeaderAttrib));//header attr

if (HeaderAttrib.TypeID == 0xffffffff) break;//last attr

if (HeaderAttrib.TypeID == 0x30)//name attribut

{

WORD FileAttrOffset = AttrOffset + HeaderAttrib.OffsetContent;

memcpy(&NameAttr, buf + FileAttrOffset, sizeof(NameAttr));

if (NameAttr.NameLength > 0)

{

char Name[30]; int j = 0;

for (int i = 0; i < NameAttr.NameLength \* 2; i += 2)

Name[j++] = NameAttr.FileName[i];

Name[j] = '\0';

printf("%s\n", Name);

}

}

AttrOffset += HeaderAttrib.Length;

}

BeginMFTSector.QuadPart += 1024;

}

}

system("pause");

return 0;

}

**Варианты заданий к лабораторной работе №6**

1. Вывести имена и основные характеристики атрибутов метафайла . (корневой каталог)
2. Вывести имена и основные характеристики атрибутов метафайла $Bitmap
3. Вывести имена и основные характеристики атрибутов метафайла $MFT
4. Вывести имена и основные характеристики атрибутов метафайла $AttrDef
5. Вывести имена и основные характеристики атрибутов метафайла $Volume
6. Вывести имена и основные характеристики атрибутов метафайла $BadClus
7. Вывести имена и основные характеристики атрибутов метафайла $LogFile
8. Вывести имена и основные характеристики атрибутов метафайла $MFTMirr
9. Вывести имена и основные характеристики атрибутов метафайла $Boot
10. Вывести имена и основные характеристики атрибутов метафайла $Secure
11. Вывести имена и основные характеристики атрибутов метафайла $UpCase
12. Вывести имена и основные характеристики атрибутов метафайла $Extend
13. Вывести имена и основные характеристики атрибутов метафайла $Quote
14. Вывести состав корневой директории заданного диска
15. Вывести имена файлов, содержащих нерезидентные атрибуты

**Контрольные вопросы**

1. Назначение MFT, определение ее местоположения.
2. Структура записи таблицы MFT.
3. Резидентные и нерезидентные атрибуты.
4. Назначение метафайлов.

### Лабораторная работа № 7

**Системный Реестр в WINDOWS NT.**

***Цель работы:*** изучить структуру системного Реестра и основные функции работы с системным Реестром.

***Теоретические сведения***

Системный Реестр в WINDOWS NT является иерархической структурой, состоящей из разделов(key) и пар параметров(name-value pair). Эта структура используется как база данных, централизованно хранящая информацию о пользователях, приложениях, операционной системе и о конфигурации компьютера. Разделы HKEY - это средства доступа к разделам Реестра самого верхнего уровня. Они определяются в API Реестра и являются точкой входа для разработчиков к остальному дереву Реестра. Пара «параметр - значение» состоит из трех частей: имени параметра, значения параметра и типа данных, т.е. типа значения. Основные три типа данных: двойное слово (DWORD - тип REG\_DWORD), строка (STRING - тип REG\_SZ) и двоичное значение (BINARY - тип REG\_BINARY).

***Некоторые действия, реализуемые изменением реестра:***

* Узел HkLM\ SYSTEM\ CurrentControlSet\ Control…

*Отключение коротких имен 8.3 в NTFS:*

…\ FileSystem\NtfsDisable8dot3NameCreation :=1

*Изменение установленных по умолчанию переменных окружения*:

... \ SessionManager\ Environment

*Очистка файла подкачки Pagefile.sys*.

... \ SessionManager\ MemoryManagement\ ClearPageFileAtShutdown :=1

* Узел HkLM\ SOFTWARE\ Microsoft\ WindowsNT\ CurrentVersion\ WinLogon ...

*Запрет перезагрузки и выключения компьютера.*

... \ ShutdownWithoutLogon :=1

*Уничтожение информации о последнем пользователе:*

... \ DontDisplayLastUserName :=1

*Потенциальные места расположения троянских программ.*

... \ System - запуск от имени системы

... \ Userinit - запуск от имени пользователя.

HkLM\ Software\ Microsoft\ Windows\ CurrentVersion\ Run\

*Снятие,установка Пароля Экранных Заставок, в ключе:*

* HkU\ Default\ ControlPanel\ Desktop\ ScreenSaveIsSecure :=1

*Переименование-Удаление Корзины*

* HkCR\ CLSID\ {645FF040-5... }\ ShellFolder\ Attributes :=70010020

**Твики реестра Windows**

**Твики реестра Windows** (англ. *tweaks* - настройки) - это настройки программного обеспечения и операционной системы, хранящиеся в системном реестре. Твики реестра реализуются при помощи **REG-файлов** - файлов, которые при их запуске автоматически вносят необходимые изменения в системный реестр Windows. Результат такой же, как и при ручном редактировании реестра через редактор. Нужные REG-файлы можно создавать самостоятельно или же использовать уже готовые, созданные другими пользователями. При этом, REG-файл может изменять как один параметр реестра, так и целые их группы.

По сути, REG-файл – это самый обычный текстовый файл с [расширением](http://www.chaynikam.info/tip_raschirenie_fayla.html) .reg.

Теперь если сделать двойной щелчок мышки по сохраненному файлу, компьютер «спросит», действительно ли нужно добавить информацию из него в реестр. Обратите внимание, что данные эти должны иметь строго определенную структуру. В противном случае, REG-файл работать не будет.

**Структура REG-файла**

В первой строке файла указывается информация о том, для редактирования какой версии Windows он предназначается. Если это Windows 98 или Windows NT, то в первой строке вписываем **«REGEDIT4»**. Для более поздних версий Windows (2000, XP, 7 и др.) – **«Windows Registry Editor Version 5.00»** (без кавычек, точно как указано - со всеми пробелами, большими буквами и т.д.). Ничего больше в этой строке быть не должно.

Вторая строка обязательно должна быть пустой.

В третьей строке в квадратных скобках […] прописывается ветка (раздел) реестра, в которой находятся изменяемые параметры.

В четвертой и последующих строках прописываются изменяемые ключи, которые находятся в указанном разделе, а также проводимые с ними действия (каждый изменяемый параметр в отдельной строке).

Формат этих строк зависит от вида изменяемых ключей:

|  |  |  |
| --- | --- | --- |
| **Двоичные параметры** | | |
| **Действие** | **Общий вид** | **Пояснения** |
| Создание | "Название"=hex:00,00,00 | Будет создан двоичный параметр с названием «Название» и значением 00 00 00. Слово hex указывает на то, что будет создан именно двоичный параметр. В REG-файлах значение для двоичных параметров указывается шестнадцатеричными числами, разделенными запятыми. Слово hex пишется маленькими буквами, двоеточие обязательно, без пробелов. |
| Изменение | "Название"=hex:00,00,00 | Существующему двоичному параметру с названием «Название» будет задано значение 00 00 00 (см. комментарий в предыдущем пункте) |
| Удаление | "Название"=- | Существующий двоичный параметр с названием «Название» будет удален из реестра. |
| **Параметры DWORD** | | |
| Создание | "Название"=dword:00000000 | Будет создан параметр DWORD с названием «Название» и значением 00000000. Большинство параметров DWORD имеют значение либо 1, либо 0. В REG-файлах значение для DWORD указываются в шестнадцатеричном формате. Поэтому 0 будет иметь вид 00000000, а 1 – 00000001. Слово dword пишется маленькими буквами, двоеточие обязательно, без пробелов. |
| Изменение | "Название"=dword:00000000 | Существующему параметру DWORD с названием «Название» будет задано значение 00000000 (см. комментарий к предыдущему пункту) |
| Удаление | "Название"=- | Существующий DWORD параметр с названием «Название» будет удален из реестра. |
| Если заметили, строка для удаления DWORD параметра ничем не отличается от строки удаление двоичного параметра. Дело в том, что в одном разделе реестра не может существовать несколько параметров с одинаковыми названиями, даже если эти параметры будут разных видов. Не трудно догадаться, что строка для удаления параметров остальных видов будет иметь такой же вид. | | |
| **Строковые параметры** | | |
| Создание | "Название"="Значение" | Будет создан строковый параметр с названием «Название» и значением «Значение». Обратите внимание: кавычки обязательны до и после знака «=». Без пробелов. |
| Изменение | "Название"="Значение" | Существующему строковому параметру с названием «Название» будет задано новое значение «Значение». |
| Удаление | "Название"=- | Существующий строковый параметр с названием «Название» будет удален из реестра. |

Об остальных видах параметров реестра говорить не будем, поскольку пользователями они редактируются редко.

Последняя строка REG-файла **обязательно должна быть пустой**.

Рассмотрим пример:

|  |
| --- |
| Windows Registry Editor Version 5.00  [HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SystemRestore] "1"=dword:00000000 "11"="2" "111"=hex:00,00,00 "1111"=- |

**Этот REG-файл** в разделе реестра «HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SystemRestore» **создаст**:

• Параметр DWORD с названием «1» и значением «0» (если в разделе уже существует параметр DWORD с названием «1», то его значение станет «0»)

• Строковый параметр с названием «11» и значением «2» (если в разделе уже существует строковый параметр с названием «11», то его значение станет «2»)

• Двоичный параметр с названием «111» и значением 00 00 00 (если в разделе уже существует двоичный параметр с названием «111», то его значение станет «00 00 00»)

• Удалит из раздела параметр (какого бы вида он ни был) с названием «1111».

А что же делать, если кроме этого нужно еще изменить параметры, находящиеся в другом разделе реестра? Все очень просто. Оставляем одну пустую строку после последнего параметра, после чего в квадратных скобках […] указываем новый раздел, а в последующих строках - параметры, которые в нем нужно изменить.

Добавим к нашему примеру еще несколько строк:

|  |
| --- |
| Windows Registry Editor Version 5.00  [HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SystemRestore] "1"=dword:00000000 "11"="2" "111"=hex:00,00,00 "1111"=-  [HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion] "2"="ххх" "22"=- |

Такой REG-файл, кроме изложенных выше действий, в разделе «HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion» создаст строковый параметр с названием «2» и значением «ХХХ» и удалит параметр с названием «22».

Последняя строка в таком REG-файле тоже обязательно должна быть пустой.

**Функции WIN32 API для работы с системным Реестром.**

Функция **RegOpenKeyEx** - открытие раздела. При успешном открытии раздела функцией возвращается ERROR\_SUCCESS , в противном случае - код ошибки:

LONG RegOpenKeyEx ( HKEY hKey, // дескриптор корневого раздела LPCTSTR lpSubKey, // указатель на адрес подраздела корневого раздела, в котором создается подраздел . DWORD ulOptions, //зарезервирован и равен 0. REGSAM samDesired, //доступ к разделу. PHKEY phkResult //адрес дескриптора открываемого ключа. ); .

Параметры: HKey - идентифицирует корневой ключ или один из ниже перечисленных : HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_USER. HKEY\_LOCAL\_MACHINE, HKEY\_USERS. samDesired - определяет доступ к разделу.

PhkResult - обработчик раздела приложения, т. е. дескриптор открытого ключа. Пример написания функции: RegOpenKeyEx(HKEY\_CURRENT\_USER,"SOFTWARE",0,KEY\_READ | KEY\_WRITE,&hKey); . Эта функция открывает в корневом разделе HKEY\_CURRENT\_ USER подраздел SOFTWARE с разрешениями на создание подразделов, на запись данных подраздела, на получение данных подраздела, на перечисление подразделов, на извещение об изменении.

Функция **RegCreateKeyEx**  создает новый подраздел внутри родительского. При успешном создании раздела функцией возвращается ERROR\_SUCCESS , в противном случае - код ошибки. При создании нового подраздела необходимо сначала открыть его родительский раздел функцией RegOpenKeyEx. Функция RegCreateKeyEx открывает раздел, если он существует, или создает его с новыми атрибутами:

LONG RegCreateKeyEx( HKEY hKey, // дескриптор открытого ключа.

LPCTSTR lpSubKey, // адрес открываемого ключа не может быть равен NULL.

DWORD Reserved, // зарезервирован.

LPTSTR lpClass, // адрес имя класса или типа объекта.

DWORD dwOptions, // специальный тип раздела

REGSAM samDesired, // определяет маску доступа к ключу

LPSECURITY\_ATTRIBUTES lpSecurityAttributes, //указатель на структуру LPSECURITY\_ATTRIBUTES PHKEY phkResult, //адрес дескриптора созданного ключа

LPDWORD lpdwDisposition //указатель на буфер, eсли ключ создан, то в поле определяемое LpdwDisposition, записывается значение REG\_CREATED\_NEW\_KEY, а если ключ существовал и был открыт, - значение REG\_OPENED\_EXISTING\_KEY.

Пример записи функции: RegCreatKeyEx (hKey,"MyKey", 0, "", REG\_OPTION\_NON\_VOLATILE,KEY\_ALL\_ACCESS, NULL,&hMyKey, lpdwDisposition).

Функция **RegSetValueEx**  устанавливает значение раздела в реестре. При использовании этой функции необходимо открыть раздел, значение которого необходимо установить. Эта функция позволяет создавать несколько пар параметр- значение в одном разделе. При успешной установке функцией возвращается ERROR\_SUCCESS , в противном случае - код ошибки:

LONG RegSetValueEx( HKEY hKey, // дескриптор ключа, к которому добавляются данные. LPCTSTR lpValueName,//-указатель на строку, содержащую имя добавляемых данных.

DWORD Reserved, // зарезервирован

DWORD dwType, // определяет тип информации, который будет сохранен в качестве данных. CONST BYTE \*lpData, // указатель непосредственно на данные, которые будут сохранены.

DWORD cbData // размер данных, на который указывает предыдущий аргумент. ); .

Пример записи функции: RegSetValueEx((hMyKey, "m\_name", 0, REG\_SZ, (const unsigned char\*)(LPCTSTR) lpszKeyValue, lpszKeyValue.GetLength()) .

Функция **RegCloseKey** закрывает раздел перед сохранением системных ресурсов:

LONG RegCloseKey( HKEY hKey //дескриптор закрываемого раздела, возвращенный функцией RegOpenKeyEx. ); .

Пример записи функции:

RegCloseKey(hMyKey) -закрывает подраздел My Key. При успешном закрытии раздела функцией возвращается ERROR\_SUCCESS , в противном случае - код ошибки.

Функция **RegFlushKey** записывает изменения, сделанные в Реестре на диск. Необходимость использования RegFlushKey возникает, когда нужна гарантия записи изменения в Реестре перед выполнением остального кода. При успешном выполнении функцией возвращается ERROR\_SUCCESS, в противном случае - код ошибки.

LONG RegFlushKey ( HKEY hKey //дескриптор ключа для записи. ); .

Пример записи функции: RegFlushKey(hMyKey).

Функция **RegDeleteKey** удаляет раздел вместе с его значениями, если в разделе существуют подразделы, его удалить нельзя. Нельзя удалить подраздел, если его родительский раздел был открыт функцией RegCreateKeyEx и не был закрыт функцией RegCloseKey:

LONG RegDeleteKey ( HKEY hKey, // дескриптор открытого ключа.

LPCTSTR lpSubKey // указатель на подстроку с именем удаляемого подраздела. ); .

Существует два способа удаления раздела: 1. В качестве первого параметра указать корневой раздел, во втором параметре - полный путь к удаляемому подразделу. 2. Открыть родительский раздел с помощью процедуры RegOpenKeyEx и затем передать его дескриптор в качестве первого параметра. В качестве второго параметра указывается имя дочернего раздела. Пример написания функции: RegDeleteKey (HKEY\_CURRENT\_USER,"SOFTWARE\\My Key").

***Требования к программе***

В программе должно быть реализовано:

1. Создание подраздела (название подраздела –должно совпадать с Вашей фамилией) в системном Реестре в разделе HKEY\_CURRENT\_USER в подразделе SOFTWARE и добавление параметра, содержащего текущую дату;

2. Изменение параметров подраздела в соответствии с вариантом;

3. Прочитать значение параметра из раздела HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION\System в соответствии с вариантом и объяснить его назначение ;

4. Окончание работы программы с сохранением изменений в системном Реестре; окончание работы программы без сохранения изменений в системном Реестре.

5. Для контроля работы программы необходимо вызвать программу Regedit, которая позволяет просматривать и редактировать Реестр.

6. п.1,2,3 выполнить также написанием .REG файла.

Варианты

|  |  |  |
| --- | --- | --- |
| № | *Прочитать значение параметра* | *Изменить значение параметра* |
| 1 | SystemBiosVersion (System) | Переименовать корзину |
| 2 | SystemVersion (BIOS) | Удалить инф. о последнем пользователе |
| 3 | BIOSVendor | В переменную окружения path добавить путь к своей папке |
| 4 | BIOSVersion | Добавить переменную окружения со своей фамилией |
| 5 | BIOSReleaseDate | Отключить короткие имена |
| 6 | BaseBoardManufacturer | Очистить файл подкачки |
| 7 | SystemManufacturer | Снять пароль после экранной заставки |
| 8 | VendorIdentifier (CentralProcessor/0) | Установит пароль после экранной заставки |
| 9 | ProcessorNameString | Удалить значение переменной окружения ТМР |
| 10 | Identifier | Добавить переменную окружения с сегодняшней датой |
| 11 | ~MHz | Включить короткие имена |
| 12 | SerialNumber (SCSI…) | Запретить перезагрузку компьютера |
| 13 | SystemManufacturer (BIOS) | Очистить файл подкачки |
| 14 | BIOSReleaseDate (BIOS) | Удалить значение переменной окружения ТЕМР |
| 15 | ProcessorNameString | Добавить переменную окружения бинарного типа, равную 10. |

***Контрольные вопросы***

Системный Реестр. Основные функции программы RegEdit. API функции изменения, удаления, создания подраздела в системном Реестре. Сохранение изменения в системном Реестре после окончания работы приложения.

## Лабораторная работа № 8

**Процессы**

***Цель работы:*** Научиться создавать процессы и передавать в них параметры

***Теоретические сведения***

В Windows есть возможность запустить одну программу из другой. При этом запускающая программа называется процессом-родителем, а запускаемая - процессом-потомком.

Описание

BOOL CreateProcess

(

LPCTSTR lpApplicationName, *// имя исполняемого модуля*

LPTSTR lpCommandLine, *// Командная строка*

LPSECURITY\_ATTRIBUTES lpProcessAttributes, *// Указатель на структуру SECURITY\_ATTRIBUTES*

LPSECURITY\_ATTRIBUTES lpThreadAttributes, *// Указатель на структуру SECURITY\_ATTRIBUTES*

BOOL bInheritHandles, *// Флаг наследования текущего процесса*

DWORD dwCreationFlags, *// Флаги способов создания процесса*

LPVOID lpEnvironment, *// Указатель на блок среды*

LPCTSTR lpCurrentDirectory, *// Текущий диск или каталог*

LPSTARTUPINFO lpStartupInfo, *// Указатель нас структуру STARTUPINFO*

LPPROCESS\_INFORMATION lpProcessInformation *// Указатель нас структуру PROCESS\_INFORMATION*

);

**lpApplicationName**. Указатель на строку которая заканчивается нулем и содержит имя выполняемого модуля. Этот параметр может быть **NULL** тогда имя модуля должно быть в **lpCommandLine** самым первым элементом. Если операционная система **NT** и модуль **16** разрядов этот параметр **NULL** обязательно. имя модуля может быть абсолютным или относительным. Если относительное то будет использована информация из **lpCurrentDirectory** или текущий каталог.

**lpCommandLine**.Командная строка. Здесь передаются параметры. Она может быть **NULL**. Здесь можно указать и путь и имя модуля.

**lpProcessAttributes**.Здесь определяются атрибуты защиты для нового приложения. Если указать **NULL** то система сделает это по умолчанию.

**lpThreadAttributes**. Здесь определяются атрибуты защиты для первого потока созданного приложением. **NULL** опять приводит к установке по умолчанию.

**bInheritHandles**. Флаг наследования от процесса производящего запуск. Здесь наследуются дескрипторы. Унаследованные дескрипторы имеют те же значения и права доступа, что и оригиналы.

**dwCreationFlags**. Флаг способа создание процесса и его приоритет.

CREATE\_DEFAULT\_ERROR\_MODE *Новый процесс не наследует режим ошибок (error mode) вызывающего процесса.*

CREATE\_NEW\_CONSOLE *Новый процесс получает новую консоль вместо того, чтобы унаследовать родительскую.*

CREATE\_NEW\_PROCESS\_GROUP *Создаваемый процесс - корневой процесс новой группы.*

CREATE\_SEPARATE\_WOW\_VDM *только Windows NT: Если этот флаг установлен, новый процесс запускается*

*в собственной Virtual DOS Machine (VDM).*

CREATE\_SHARED\_WOW\_VDM *только Windows NT: Этот флаг указывает функции CreateProcess запустит*

*новый процесс в разделяемой Virtual DOS Machine.*

CREATE\_SUSPENDED *Первичная нить процесса создается в спящем (suspended) состоянии*

*и не выполняется до вызова функции ResumeThread.*

CREATE\_UNICODE\_ENVIRONMENT *Если этот флаг установлен, блок переменных окружения, указанный в*

*параметре lpEnvironment, использует кодировку Unicode. Иначе - кодировку ANSI.*

DEBUG\_PROCESS *Если этот флаг установлен, вызывающий процесс считается отладчиком,*

*а новый процесс - отлаживаемым.*

DEBUG\_ONLY\_THIS\_PROCESS *Если этот флаг не установлен и вызывающий процесс находится под отладкой,*

*новый процесс так же становится отлаживаемым тем же отладчиком.*

DETACHED\_PROCESS *Создаваемый процесс не имеет доступа к родительской консоли.*

*Этот флаг нельзя использовать с флагом CREATE\_NEW\_CONSOLE.*

HIGH\_PRIORITY\_CLASS *Указывает на то, что процесс выполняет критичные по времени задачи*

IDLE\_PRIORITY\_CLASS *Указывает процесс, выполняются только когда система находится в состоянии ожидания*

NORMAL\_PRIORITY\_CLASS *Указывает на процесс, без каких либо специальных требований к выполнению.*

REALTIME\_PRIORITY\_CLASS *Указывает процесс имеющий наивысший возможный приоритет.*

**lpEnvironment**. Указывает на блок среды. Если **NULL**, то будет использован блок среды родительского процесса. Блок среды это список переменных **имя=значение** в виде строк с нулевым окончанием.

**lpCurrentDirectory**.Указывает текущий диск и каталог. Если **NULL** то будет использован диск и каталог процесса родителя.

**lpStartupInfo** Используется для настройки свойств процесса, например расположения окон и заголовок. Структура должна быть правильно инициализирована

STARTUPINFO sti; *// структура*

ZeroMemory(&sti,sizeof(STARTUPINFO)); *// обнулить*

sti.cb=sizeof(STARTUPINFO); *// указать размер*

**lpProcessInformation** Структура **PROCESS\_INFORMATION** с информацией о процессе. Будет заполнена **Windows**.

***Требования к программе***

Входные данные для процесса-потомка вводятся с клавиатуры или из командной строки процесса-родителя (по варианту) и передаются в потомок через командную строку.

**Пример программы-потомка**

#include <string.h>

#include <stdio.h>

#include <Windows.h>

int main(int c, char \*argv[])

{

if (c == 3)

{

int a = atoi(argv[1]);

int b = atoi(argv[2]);

printf("%s + %s = %d\n", argv[1], argv[2], a + b);

system("pause");

return 0;

}

else

{

printf("no parameters"); system("pause");

return -1;

}

}

**Пример программы-родителя**

#include <Windows.h>

#include <stdio.h>

int main(){

STARTUPINFO cif;

DWORD code; int i;

ZeroMemory(&cif, sizeof(STARTUPINFO));

PROCESS\_INFORMATION pi;

LPSTR s=GetCommandLine();// лучше парaметры main

bool result = CreateProcess("c:\\Projects\\child\\Debug\\child.exe", s, NULL, NULL, false, NULL, NULL, "c:\\Projects\\child\\Debug", &cif, &pi);

if (result == false)

printf("error");

else

{

WaitForSingleObject(pi.hProcess, INFINITE);

printf("child end \n");

}

system("pause");

return 0;

}

**Варианты**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| №  вар. | Способ запуска потомка | Задание параметров  родителем | Окно запуска потомка | Функция-потомок |
| 1 | Параллельно с родителем | Ком.строка | В отдельном окне | y = a\*b\*c + d |
| 2 | Родитель ожидает выполнения потомка | клавиатура | В окне родителя | y = a/b + x\*y |
| 3 | Параллельно с родителем | клавиатура | В отдельном окне | z = x + 5/y |
| 4 | Родитель ожидает выполнения потомка | Ком.строка | В окне родителя | m = 15\*n + x\*y |
| 5 | Параллельно с родителем | клавиатура | В окне родителя | y = x + a\*b - c |
| 6 | Родитель ожидает выполнения потомка | Ком.строка | В отдельном окне | z = a/b + c/d |
| 7 | Параллельно с родителем | Ком.строка | В окне родителя | f = B\*sinX+ A\*cosX |
| 8 | Родитель ожидает выполнения потомка | клавиатура | В отдельном окне | y = |sinX| + c\*d |
| 9 | Параллельно с родителем | клавиатура | В окне родителя | z = cosX + |a\*b| |
| 10 | Родитель ожидает выполнения потомка | Ком.строка | В окне родителя | y = sinX - 1/cosX |
| 11 | Параллельно с родителем | Ком.строка | В отдельном окне | p = a\*b/c + |x| |
| 12 | Родитель ожидает выполнения потомка | клавиатура | В отдельном окне | m = n\*X + sinX |
| 13 | Параллельно с родителем | клавиатура | В отдельном окне | a = b - c\*d/|b| |
| 14 | Родитель ожидает выполнения потомка | Ком.строка | В окне родителя | p = sinX\*cosX + 1 |
| 15 | Параллельно с родителем | клавиатура | В отдельном окне | y = sin|X| + cos|Z| |
| 16 | Родитель ожидает выполнения потомка | Ком.строка | В окне родителя | x = d\*sin|B| + c |
| 17 | Параллельно с родителем | клавиатура | В отдельном окне | y = b\*cos|X| - 16 |
| 18 | Родитель ожидает выполнения потомка | клавиатура | В окне родителя | z = a\*b\*sinX |
| 19 | Параллельно с родителем | Ком.строка | В окне родителя | y = a\*b\*c + sin|X| |
| 20 | Родитель ожидает выполнения потомка | клавиатура | В отдельном окне | m = |x\*Y| + z\*sinY |
| 21 | Параллельно с родителем | Ком.строка | В окне родителя | n = a - b - c\*sin|X) |
| 22 | Родитель ожидает выполнения потомка | клавиатура | В отдельном окне | y = a\*b\*c\*d + 19 |
| 23 | Параллельно с родителем | Ком.строка | В окне родителя | x = c\*d + |a\*b| |
| 24 | Родитель ожидает выполнения потомка | Ком.строка | В окне родителя | z = cosX + |a\*b| |
| 25 | Параллельно с родителем | клавиатура | В отдельном окне | z = x - y/10 |

### Лабораторная работа № 9

**Разработка многопоточного приложения.**

**Цель работы**. Исследовать возможности многопоточности, а также вопросы создания и использования волокон на одно- и многоядерном процессорах.

**Теоретические сведения**

Процесс состоит из потоков. При инициализации процесса система создает всегда первичный поток. Поток определяет последовательность исполнения кода в процессе. В то время как приложение предназначено для выполнения всей задачи, поток – это мини-программа, принадлежащая приложению и выполняющая частную задачу. При параллельном выполнении потоков и процессов могут возникнуть проблемы с доступом к разделяемым ресурсам. Чтобы избежать подобных конфликтов, в Windows предусмотрены средства синхронизации потоков.

Основные функции WIN32 API для работы с потоками.

Функция создания потока CreateThread:

HANDLE CreateThread ( LPSECURITY\_ATTRIBUTES lpThreadAttributes, // указатель на атрибуты защиты нового потока

SECURITY\_ATTRIBUTES;

DWORD dwStackSize,// размер стека, выделяемого потоку ; если размер равен нулю, то поток будет иметь стек такого же размера, как и породивший его поток;

LPTHREAD\_START\_ROUTINE lpStartAddress, // адрес функции потока, с которой должен начать работу создаваемый поток;

эта функция потока должна иметь следующий прототип:

DWORD WINAPI ThreadFunc(LPVOID lpParameter);

LPVOID lpParameter, // 32 – битный параметр (или 32-битный указатель), который передает функции потока какое-либо инициализирующее значение;

DWORD dwCreationFlags, // Если этот аргумент равен нулю, то выполнение этого потока начнется немедленно. Если этот аргумент будет равен CREATE\_SUSPENDED, то начало выполнения потока будет задержано до определенных событий( например, до вызова функции ResumeThread();

LPDWORD lpThreadId // адрес идентификатора созданного потока. ); . Если функция завершилась без ошибок, она возвращает значение вновь созданного описателя, иначе значение функции равно NULL.

Функция изменения относительного приоритета потока в пределах одного процесса :

BOOL SetThreadPriority(HANDLE hThread, int nPriority), где hThread – описатель потока, приоритет которого изменяется, nPriority - приоритет, который принимает одно из следующих значений: THREAD\_PRIORITY\_TIME\_CRITICAL // Критичный по времени THREAD\_PRIORITY\_HIGHEST // Наивысший THREAD\_PRIORITY\_ABOVE\_NORMAL // Выше нормального THREAD\_PRIORITY\_NORMAL // Нормальный

THREAD\_PRIORITY\_BELOW\_NORMAL // Ниже нормального THREAD\_PRIORITY\_LOWEST // Самый низкий THREAD\_PRIORITY\_IDLE // Простаивающий

В момент создания потока начальное значение его относительного приоритета равно THREAD\_PRIORITY\_NORMAL.

***Синхронизация потоков***

*Критические секции*обеспечивают синхронизацию подобно мьютексам (о мьютексах см. далее) за исключением того, что объекты, представляющие критические секции, доступны в пределах одного процесса. События, мьютексы и семафоры также можно использовать в "однопроцессном" приложении, однако критические секции обеспечивают более быстрый и более эффективный механизм взаимно-исключающей синхронизации. Подобно мьютексам объект, представляющий критическую секцию, может использоваться только одним потоком в данный момент времени, что делает их крайне полезными при разграничении доступа к общим ресурсам. Трудно предположить что-нибудь о порядке, в котором потоки будут получать доступ к ресурсу, можно сказать лишь, что система будет "справедлива" ко всем потокам.

  #include <windows.h>

  #include <process.h>

  #include <stdio.h>

    CRITICAL\_SECTION cs;

  int a[ 5 ];

    void Thread( void\* pParams )

  {  int i, num = 0;

      while ( TRUE )

    {   EnterCriticalSection( &cs );

       for ( i = 0; i < 5; i++ ) a[ i ] = num;

       LeaveCriticalSection( &cs );

       num++;

    }

  }

    int main( void )

   {    InitializeCriticalSection( &cs );

    \_beginthread( Thread, 0, NULL );

    while( TRUE )

    {

       EnterCriticalSection( &cs );

       printf( "%d %d %d %d %d\n",

               a[ 0 ], a[ 1 ], a[ 2 ],

               a[ 3 ], a[ 4 ] );

       LeaveCriticalSection( &cs );

    }

    return 0;

  }

**Мьютексы (взаимоисключения)**

Мьютекс (взаимоисключение, mutex) - это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов - одновременный доступ к общему ресурсу исключается. Например, чтобы исключить запись двух потоков в общий участок памяти в одно и то же время, каждый поток ожидает, когда освободится мьютекс, становится его владельцем и только потом пишет что-либо в этот участок памяти. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу.

Два (или более) процесса могут создать мьютекс с одним и тем же именем, вызвав метод CreateMutex. Первый процесс действительно создает мьютекс, а следующие процессы получают хэндл уже существующего объекта. Это дает возможность нескольким процессам получить хэндл одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс. Если используется такой подход, желательно установить флаг *bInitialOwner* в FALSE, иначе возникнут определенные трудности при определении действительного создателя мьютекса.

Несколько процессов могут получить хэндл одного и того же мьютекса, что делает возможным взаимодействие между процессами. Вы можете использовать следующие механизмы такого подхода:

* Дочерний процесс, созданный при помощи функции CreateProcess может наследовать хэндл мьютекса в случае, если при его (мьютекса) создании функией CreateMutex был указан параметр *lpMutexAttributes*.
* Процесс может получить дубликат существующего мьютекса с помощью функции DuplicateHandle.
* Процесс может указать имя существующего мьютекса при вызове функций OpenMutex или CreateMutex.

Вообще говоря, если вы синхронизируете потоки одного процесса, более эффективным подходом является использование критических секций.

  #include <windows.h>

  #include <process.h>

  #include <stdio.h>

  HANDLE hMutex;

  int a[ 5 ];

  void Thread( void\* pParams )

  {

     int i, num = 0;

     while ( TRUE )

     {

        WaitForSingleObject( hMutex, INFINITE );

        for ( i = 0; i < 5; i++ ) a[ i ] = num;

        ReleaseMutex( hMutex );

        num++;

     }

  }

  int main( void )

  {

     hMutex = CreateMutex( NULL, FALSE, NULL );

     \_beginthread( Thread, 0, NULL );

     while( TRUE )

     {

        WaitForSingleObject( hMutex, INFINITE );

        printf( "%d %d %d %d %d\n",  a[ 0 ], a[ 1 ], a[ 2 ],  a[ 3 ], a[ 4 ] );

        ReleaseMutex( hMutex );

     }

     return 0;

  }

**События**

А что, если мы хотим, чтобы в предыдущем примере второй поток запускался каждый раз после того, как основной поток закончит печать содержимого массива, т.е. значения двух последующих строк будут отличаться строго на 1?

Событие - это объект синхронизации, состояние которого может быть установлено в сигнальное путем вызова функций SetEvent или PulseEvent. Существует два типа событий:

|  |  |
| --- | --- |
| **Тип объекта** | **Описание** |
| Событие с ручным сбросом | Это объект, сигнальное состояние которого сохраняется до ручного сброса функцией ResetEvent. Как только состояние объекта установлено в сигнальное, все находящиеся в цикле ожидания этого объекта потоки продолжают свое выполнение (освобождаются). |
| Событие с автоматическим сбросом | Объект, сигнальное состояние которого сохраняется до тех пор, пока не будет освобожден единственный поток, после чего система автоматически устанавливает несигнальное состояние события. Если нет потоков, ожидающих этого события, объект остается в сигнальном состоянии. |

События полезны в тех случаях, когда необходимо послать сообщение потоку, сообщающее, что произошло определенное событие. Например, при асинхронных операциях ввода и вывода из одного устройства, система устанавливает событие в сигнальное состояние когда заканчивается какая-либо из этих операций. Один поток может использовать несколько различных событий в нескольких перекрывающихся операциях, а затем ожидать прихода сигнала от любого из них.

Поток может использовать функцию CreateEvent для создания объекта события. Создающий событие поток устанавливает его начальное состояние. В создающем потоке можно указать имя события. Потоки других процессов могут получить доступ к этому событию по имени, указав его в функции OpenEvent.

Поток может использовать функцию PulseEvent для установки состояния события в сигнальное и затем сбросить состояние в несигнальное после освобождения соответствующего количества ожидающих потоков. В случае объектов с ручным сбросом освобождаются все ожидающие потоки. В случае объектов с автоматическим сбросом освобождается только единственный поток, даже если этого события ожидают несколько потоков. Если ожидающих потоков нет, PulseEvent просто устанавливает состояние события в несигнальное.

  #include <windows.h>

  #include <process.h>

  #include <stdio.h>

  HANDLE hEvent1, hEvent2;

  int a[ 5 ];

  void Thread( void\* pParams )

  {

     int i, num = 0;

     while ( TRUE )

     {

        WaitForSingleObject( hEvent2, INFINITE );

        for ( i = 0; i < 5; i++ ) a[ i ] = num;

        SetEvent( hEvent1 );

        num++;

     }

  }

  int main( void )

  {

     hEvent1 = CreateEvent( NULL, FALSE, TRUE, NULL );

     hEvent2 = CreateEvent( NULL, FALSE, FALSE, NULL );

     \_beginthread( Thread, 0, NULL );

     while( TRUE )

     {

        WaitForSingleObject( hEvent1, INFINITE );

        printf( "%d %d %d %d %d\n",

                a[ 0 ], a[ 1 ], a[ 2 ],

                a[ 3 ], a[ 4 ] );

        SetEvent( hEvent2 );

     }

     return 0;

  }

***Семафоры.***

Объекты ядра семафор (Semaphore) используются для учета ресурсов. Они позволяют потоку запрашивать число доступных ресурсов. Если один или более ресурсов свободны, счетчик доступных ресурсов после запроса уменьшается на 1.Система разрешает другому потоку запрашивать какой-либо ресурс только после уменьшения счетчиков доступных ресурсов. Поскольку на счетчик ресурсов семафора могут влиять несколько потоков, семафоры не передаются во владение какому-либо потоку. Один поток может ждать семафор (уменьшив его счетчик ресурсов), а другой освободить семафор (увеличив счетчик ресурсов).

Функция создания семафора.

HANDLE CreateSemaphore (LPSECURITY\_ATTRIBUTE lpsa, LONG cSemInitial,LONG cSemMax,LPTSTR lpszSemName).

Эта функция создаёт семафор, максимальное значение счетчика которого cSemMax. Параметр сSemInitial позволяет задать начальное состояние счетчика. Последний параметр функции lpszSemName - указатель на имя семафора. В дальнейшем это имя позволит получить описатель семафора из других процессов с помощью CreateSemaphore или OpenSemaphore. Функция открытия семафора.

HANDLE OpenSemaphore (DWORD fdwAccess,BOOL fInherit, LPTSTR lpszName). Параметр fdwAccess может быть равен либо SYNCHRONIZE, либо SEMAPHORE\_ALL\_ACCESS. Параметр fInherit определяет, унаследует ли дочерний процесс данный описатель данного объекта. Параметр lpszName -это указатель на объект в виде строки с нулевым символом в конце.

Функция перевода семафора в свободное состояние.

BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG cRelease, LPLONG lplPrevious).

Любой поток может вызвать эту функцию когда угодно, поскольку объекты семафор не принадлежат лишь какому-то одному потоку. С её помощью счетчик ресурсов можно увеличивать более чем на 1 одновременно. Параметр cRelease определяет, как должен освобождаться семафор. Последний параметр функции ReleaseSemaphore - lplPrevious - указатель на переменную типа LONG, в которой возвращается значение счетчика ресурсов, предшествующее тому, что получилось после его увеличения на cRelease. Если Вас не интересует это значение, передайте в параметре значение NULL.

**Волокна**

Потоки в Windows реализуются на уровне ядра операционной системы, которое отлично осведомлено об их существовании и «коммутирует» их в соответствии с созданным Майкрософт алгоритмом. В то же время волокна реализованы на уровне кода пользовательского режима, ядро ничего не знает о них, и процессорное время распределяется между волокнами по алгоритму, определяемому нами. А раз так, то о вытеснении волокон говорить не приходится — по крайней мере, когда дело касается ядра.

В потоке может быть одно или несколько волокон. Для ядра поток — все то, что можно вытеснить и что выполняет код. Единовременно поток будет выполнять код лишь одного волокна — какого, решает сам программист. Приступая к работе с волокнами, прежде всего необходимо преобразовать существующий поток в волокно. Это делает функция ConvertThreadToFiber:

PV0ID ConvertThreadToFiber(PVOID pvParam);

Она создает в памяти контекст волокна (размером около 200 байтов). В него входят следующие элементы:

■ определенное программистом значение; оно получает значение параметра pvParam, передаваемого в ConvertThreadToFiber;

■ заголовок цепочки структурной обработки исключения;

■ начальный и конечный адреса стека волокна; при преобразовании потока в волокно оп служит и стеком потока;

■ регистры процессора, включая указатели стека и команд.

Создав и инициализировав контекст волокна, программист сопоставляет его адрес с потоком, преобразованным в волокно, и теперь оно выполняется в этом потоке. ConvertThreadToFiber возвращает адрес, по которому расположен контекст волокна. Этот адрес еще понадобится вам, но ни считывать, ни записывать по нему напрямую ни в коем случае нельзя — с содержимым этой структуры работают только функции, управляющие волокнами. При вызове ExitThread завершаются и волокно, и поток.

Нет смысла преобразовывать поток в волокно, если вы не собираетесь создавать дополнительные волокна в том же потоке. Чтобы создать другое волокно, поток (выполняющий в данный момент волокно), вызывается функция CreateFiber:

PVOID CreateFiber(

DW0RD dwStackSize,

PFIBER\_START\_ROUTINE pfnStartAddress,

PVOID pvParam);

Сначала она пытается создать новый стек, размер которого задан в параметре dwStackSize. Обычно передают 0, и тогда максимальный размер стека ограничивается 1 Мб, а изначально ему передается две страницы памяти. Если вы укажете ненулевое значение, то для стека будет зарезервирован и передан именно такой объем памяти.

Функция CreateFiber создает и инициализирует новую структуру, представляющую контекст исполнения волокна. При этом пользовательское значение устанавливается по значению параметра pvParam, сохраняются начальный и конечный адреса памяти нового стека, а также адрес функции волокна (переданный в параметре pfnStartAddress).

Аргумент pfnStartAddress задает адрес функции волокна, которую программист реализовывает самостоятельно. Эта функция должна соответствовать следующему прототипу:

VOID WINAPI FiberFunc(PVOID pvParam);

Когда волокно получает процессорное время в первый раз, эта функция вызывается и получает значение pvParam, исходно переданное функции CreateFiber. В этой функции вы можете делать что угодно, но в прототипе тип возвращаемого ею значения определен как VOID — не потому, что это значение бессмысленно, а просто потому, что функция волокна не должна завершаться, пока существует волокно! Как только функция волокна завершится, поток и все созданные в нем волокна тут же будут уничтожены.

Подобно ConvertThreadToFiber, функция CreateFiber возвращает адрес контекста исполнения волокна. Но, в отличие от ConvertThreadToFiber, исполнение созданного функцией CreateFiber волокна не начинается, пока исполняется текущее волокно. Дело в том, что исполняться может только одно волокно потока одновременно. Чтобы запустить новое волокно, вызывается функция SwitchToFiber.

VOID SwitchToFiber(PVOID pvFiberExecutionContext);

Эта функция принимает единственный параметр (pvFiberExecutionContext) — адрес контекста волокна, полученный в предшествующем вызове ConvertThreadToFiber или CreateFiber. По этому адресу она определяет, какому волокну предоставить процессорное время. SwitchToFiber осуществляет такие операции:

1. Сохраняет в контексте выполняемого в данный момент волокна ряд текущих регистров процессора, включая указатели команд и стека.

2. Загружает в регистры процессора значения, ранее сохраненные в контексте волокна, подлежащего выполнению. В их число входит указатель стека, и поэтому при переключении на другое волокно используется именно его стек.

3. Связывает контекст волокна с потоком, и тот выполняет указанное волокно.

4. Восстанавливает указатель команд. Поток (волокно) продолжает выполнение с того места, на каком волокно было прервано в последний раз.

Применение SwitchToFiber — единственный способ выделить волокну процессорное время. Поскольку ваш код должен явно вызывать эту функцию в нужные моменты, вы полностью управляете распределением процессорного времени для волокон. Помните: такой вид планирования не имеет ничего общего с планированием потоков. Поток, в рамках которого выполняются волокна, всегда может быть вытеснен операционной системой. Когда поток получает процессорное время, выполняется только выбранное волокно, и никакое другое не получит управление, нока вы сами не вызовете SwitchToFiber.

Для уничтожения волокна предназначена функция DeleteFiber:

VOID DeleteFiber(PVOID pvFiberExecutionContext);

Она удаляет волокно, чей адрес контекста определяется параметром pvFiberExecutionContext, освобождает память, занятую стеком волокна, и уничтожает его контекст. Но, если передать адрес волокна, связанного в данный момент с потоком, DeleteFiber сама вызывает ExitThread — в результате поток и все созданные в нем волокна «погибают».

DeleteFiber обычно вызывается волокном, чтобы удалить другое волокно. Стек удаляемого волокна уничтожается, а его контекст освобождается.

И вот еще одна разница между волокнами и потоками: потоки, как правило, уничтожают себя сами, обращаясь к ExitThread. Использование с этой целью TerminateThread считается плохим тоном — ведь тогда система не уничтожает стек потока. Удаление волокна, занятого пересчетом, — операция вполне допустимая, стек волокна и его контекст корректно уничтожаются. Если использовать потоки, а не волокна, интерфейсный поток не смог бы корректно уничтожить поток, занятый пересчетом, - пришлось бы задействовать какой-нибудь механизм межпоточного взаимодействия и ждать, пока поток пересчета не завершится сам.

После удаления всех волокон также можно удалить их состояние из потока, исходного вызвавшего ConvertThreadToFiber, с помощью ConvertFiberToThread, — так удастся полностью освободить память, использованную для преобразования потока в волокно.

**Пример программы с 2 потоками**

#include <windows.h>

#include <process.h>

#include <stdio.h>

CRITICAL\_SECTION cs;

int a[5];

DWORD WINAPI ThreadFunc1(LPVOID lpParameter)

//void Thread(void\* pParams)

{

int i, num = 0;

//\_\_rdtsc();

while (TRUE)

{

EnterCriticalSection(&cs);

for (i = 0; i < 5; i++) a[i] = num;

LeaveCriticalSection(&cs);

num++;

}

}

DWORD WINAPI ThreadFunc2(LPVOID lpParameter)

{

int i;

while (TRUE){

EnterCriticalSection(&cs);

for (i = 0; i < 5; i++) a[i] = 1;

LeaveCriticalSection(&cs);

}

}

int main(void)

{

char c;

DWORD ThreadID, Thread2ID;

InitializeCriticalSection(&cs);

HANDLE h1=CreateThread(nullptr, 0, ThreadFunc1, nullptr, CREATE\_SUSPENDED, &ThreadID);

SetThreadAffinityMask(h1, 0x00000002);

SetThreadPriority(h1, THREAD\_PRIORITY\_ABOVE\_NORMAL);

HANDLE h2 = CreateThread(nullptr, 0, ThreadFunc2, nullptr, CREATE\_SUSPENDED, &Thread2ID);

SetThreadPriority(h1, THREAD\_PRIORITY\_ABOVE\_NORMAL);

SetThreadAffinityMask(h2, 0x00000004);

ResumeThread(h1);

ResumeThread(h2);

//\_beginthread(ThreadFunc, 0, NULL);

while (TRUE)

{

EnterCriticalSection(&cs);

printf("%d %d %d %d %d\n",a[0], a[1], a[2],a[3], a[4]);

LeaveCriticalSection(&cs);

}

return 0;

}

**Варианты заданий**

1. Исследовать работу 2 потоков на 1 ядре. Тип синхронизации – критичные секции
2. Исследовать работу 2 потоков на 2 ядрах. Тип синхронизации - мьютексы
3. Исследовать работу 2 волокон в одном потоке. Тип синхронизации – события.
4. Исследовать работу 2 волокон на различных потоках на 1 ядре. Тип синхронизации - семафор
5. Исследовать работу 2 волокон на различных потоках на разных ядрах. Тип синхронизации – критичные секции
6. Исследовать работу 2 потоков на разных ядрах. Тип синхронизации – критичные секции.
7. Исследовать работу 2 волокон на различных потоках на 1 ядре. Тип синхронизации – критичные секции.
8. Исследовать работу 2 волокон на 1 потоке. Тип синхронизации – критичные секции
9. Исследовать работу 2 потоков на разных ядрах. Тип синхронизации – события.
10. Исследовать работу 2 потоков на 1 ядре. Тип синхронизации – мьютексы.
11. Исследовать работу 2 потоков на разных ядрах. Тип синхронизации – семафоры.
12. Исследовать работу 2 потоков на разных ядрах. Тип синхронизации – события.
13. Исследовать работу 2 волокон на различных потоках на разных ядрах. Тип синхронизации – мьютексы.
14. Исследовать работу 2 волокон на различных потоках на разных ядрах. Тип синхронизации – события.
15. Исследовать работу 2 волокон на различных потоках на 1 ядре. Тип синхронизации – события.

### Лабораторная работа № 10

**Исследование защищенного режима CPU.**

**Цель работы:** написание программы для включения 32-битного защищенного режима CPU, возврата в реальный режим и выполнение операции сложения соответствующих операндов. Организация памяти, способы адресации в защищенном режиме, методика перехода и возврата в реальный режим излагается на лекциях.

**Теоретические сведения**

При выполнении работы используется сегментная модель памяти. При этом в сегментном регистре располагается селектор, являющийся индексом в служебной структуре (таблицу) GDT.

Структура дескриптора может быть представлена в следующем виде:

pm_in5_1

где,

* двухбитовое поле RPL (Requested Privilege Level) содержит номер уровня привилегий, которое имеет текущая программа.
* бит TI (Table Indicator) определяет таблицу, из которой выбирается нужный дескриптор. Если бит TI = 0, то обращение производится к глобальной дескрипторной таблице GDT, если TI = 1 - то к текущей локальной дескрипторной таблице LDT.
* index - это собственной номер дескриптора, от 0 до 8191. Т.к. поле индекса состоит из 13 бит, то максимальное число дескрипторов, одновременно существующих в системе равно 213, т.е. 8192.

ОС должна иметь одну таблицу GDT. Таблицу используют все программы и задачи системы.GDT состоит из дескрипторов и находится в оперативной памяти (где и программы). Строить GDT нужно в программе. Дескриптор – это структура, описывающая сегмент. Сегментный дескриптор занимает 8 байт (2 двойных слова). Структура дескриптора может быть представлена в следующем виде:

Биты : Описание

0..15: предел, биты 0..15

16..31: адрес сегмента, биты 0..15

32..39: адрес сегмента, биты 16..23

40: бит A (Accessed)

41..43: Тип сегмента

44: бит S (System)

45,46: DPL (Descriptor Privilege Level)

47: бит P (Present)

48..51: предел, биты 16..19

52: бит U (User)

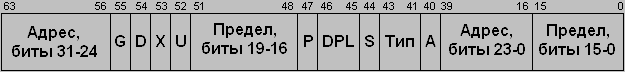
53: бит X (reserved)

54: бит D (Default size)

55: бит G (Granularity)

56..63: адрес сегмента, биты 24..31

Или графически:



Дескриптор состоит из следующих полей.

Адрес (базы) - адрес нулевого байта описываемого сегмента в 4 Гб линейном адресном пространстве (т.е. адрес, с которого начинается сегмент). Процессор образует единый 32-х битный адрес.

Лимит сегмента определяет размер сегмента. Реальный лимит сегмента зависит от бита гранулярности (**G-granularity**);

- если бит гранулярности сброшен (0), то 20-битное значение и будет тем самым лимитом сегмента, байт ;

- если бит гранулярности установлен (1), то всё 20-битное значение автоматически увеличивается в 1000h раз, т.е. при G=1 измерить в 4Кб единицах. Например, если G=1 и поле «Лимит сегмента» = 0000Fh (15 байт), то реальный лимит (размер) данного сегмента равен 0Fh\*1000h=0F000h.

Следует отметить, что если поле «Лимит сегмента» содержит значение равное 0, то это значит, что сегмент имеет размер в 1 байт (а не ноль!) при G=0, и размер в 4Кб при G=1. Т.е. сегмент никак не может иметь нулевую длину, минимум – 1 байт, максимум – 0хFFFFFh \* 4Кб = 4 Гб.

Есть еще один бит, от которого зависит смысловое значение этого поля. Бит направления роста сегмента (**B-big)**);

- если этот бит сброшен (0), то разрешены все смещения от 0 до лимита

- если установлен (1) – то все, кроме от 0 до лимита

 Тип - определяет тип сегмента, права доступа к сегменту и направление роста сегмента (бит B). Значение этого поля зависит от значения поля «Тип дескриптора» (S-descriptor type). Значение этого поля различно для разных типов сегментов (кода, данных и системного)

 S (descriptor type) – флаг «тип дескриптора». Означает: если сброшен (0), то описуемый сегмент – системный, если установлен (1) – это сегмент данных или кода.

DPL (descriptor privilege level) – уровень привилегий дескриптора, определяет уровень привилегий (от 0 до 3) сегмента.

P (segment present flag) – флаг присутствия сегмента. Если установлен - сегмент присутствует в памяти; если сброшен – нет.

G (Granularity) – флаг «гранулярности». Этот флаг влияет на лимит сегмента: если сброшен – лимит измеряется в байтах, если установлен – в 4 Кб единицах. На адрес базы этот флаг не влияет.

  Первый дескриптор в GDT не используется и называется «нулевой дескриптор» (null descriptor). НАЧАЛО таблицы GDT храниться В РЕГИСТРЕ GDTR! Регистр GDTR – это регистр, как EAX, EIP, ES. Его функция заключается в хранении фиксированного числа – НАЧАЛА ТАБЛИЦЫ GDT. Начало таблицы GDT в памяти должно быть кратно 8.

Загрузить/считать значение регистра GDTR можно командами LGDT/SGDT. По умолчанию (т.е. после нажатия на кнопку Reset или включения компа) база GDT равна нулю, а лимит – FFFFh, т.е. фактически по умолчанию выделено максимум места, под FFFFh/8 = 8191 дескрипторов (минус один, учитывая null descriptor).

Программа должна выполняться из простой операционной системы, работающей в режиме реальных адресов (MS-DOS). Если запустить программу из-под ОС, работающей в защищённом режиме (Windows), то программа не заработает, т.к. процессор уже будет работать в P-Mode и не допустит повторного входа в этот режим.

Программа выполняет следующие действия: подготовиться к переходу в P-Mode, перейти в P-Mode, сообщить в программе о переходе в P-Mode. В таблице

дескрипторов описано столько дескрипторов, сколько сегментов использует программа. В программе в таблицу включены, кроме обязательного нулевого дескриптора, всегда занимающего первое место в таблице, четыре дескриптора для сегментов данных, команд, стека и дополнительного сегмента данных, который мы наложим на видеобуфер. Вывод на экран будет выполняется прямой записью в видеопамять, для чего создана процедура.

В защищённом режиме специальная система прерываний и воспользоваться функциями операционной системы нельзя. Недоступными окажутся прерывания BIOS и IRQ.

Первой командой после перехода в защищённый режим должна быть команда дальнего перехода (far jump), в которой указан селектор дескриптора сегмента кода и смещение в этом сегменте. Однако, при переходе в защищённый режим регистр CS будет содержать сегментный адрес, который использовался в режиме реальных адресов, поэтому выполнение следующей команды должно было бы привести к генерации процессором исключения. Но этого не происходит, так как эта команда не выбирается из памяти - она уже находится в конвейере процессора. Команда дальнего перехода очистит конвейер процессора, заставит его обратится к таблице GDT, выбрать оттуда дескриптор, селектор которого указан в адресе команды и начать выборку команд со смещения, указанного в этом адресе. Приведенные в программе комментарии облегчают понимание механизма защищенного режима.

;Программа для включения защищённого режима и

;возврата в режим реальных адресов.

;---------------------------------------------------------

.386p ;для использования привилегированных команд

; 16-битный сегмент, в нем находится код для входа

; и выхода из защищенного режима

pmode segment use16

assume cs:pmode, ds:pmode, es:pmode

org 100h ; СОМ, без потери общности

main proc far

start:

;---------------------------------------------------------

; селекторы определяем как константы. У всех из них биты TI = 0 (выборкадескрипторов выполняется из GDT),

;RPL = 00B - уровень привилегий - нулевой.

Code\_selector = 8

Stack\_selector = 16

Data\_selector = 24

Screen\_selector= 32

R\_Mode\_Code= 40; Селектор дескриптора сегмента кода для возврата

; в режим реальных адресов.

R\_Mode\_Data= 48; Селектор дескриптора сегментов стека и данных.

;-------------------------------------------------------

; Сохраняем сегментные регистры, для возврата в R-Mode:

mov R\_Mode\_SS,ss

mov R\_Mode\_DS,ds

mov R\_Mode\_ES,es

mov R\_Mode\_FS,fs

mov R\_Mode\_GS,gs

; Подготавливаем адрес возврата в R-Mode:

mov R\_Mode\_segment,cs

lea ax,R\_Mode\_entry

mov R\_Mode\_offset,ax

; если работать с 32-битной - памятью открываем линию А20:

in AL,92h

or AL,2

out 92h,AL

; Очистим экран

mov ax,3 ; clear screen

int 10h

; Подготовка к переходу в защищённый режим:

mov bx,offset GDT + 8 ; Нулевой дескриптор устанавливать

; не будем - он не используется.

xor eax,eax; EAX = 0

mov edx,eax; EDX = 0

push cs

pop ax ; AX = CS = сегментный адрес текущего

; сегмента кода.

shl eax,4; EAX = физический адрес начала сегмента кода.

; Эта программа, работая в среде операционной системы

; режима реальных адресов (подразумевается, MS-DOS)

; уже имеет в IP смещение относительно

; текущего сегмента кода. Определим дескриптор

; кода для защищённого режима с таким же адресом

; сегмента кода, чтобы при переходе через команду

; дальнего перехода фактически переход произошёл

; на следующую команду.

mov dx,1024; Предел сегмента кода может быть любым,

; лишь бы он покрывал весь реально

; существующий код.

mov cl,10011000b; права доступа сегмента кода (P = 1,

; DPL = 00b, S = 1, тип = 100b, A = 0)

call set\_descriptor;Конструируем дескриптор кода,в функции

lea dx,Stack\_seg\_start ;EDX=DX=начало стека (см. метку)

add eax,edx; EAX уже содержит адрес начала сегмента

; кода, сегмент стека начнётся с последней

; метки программы Stack\_seg\_start.

mov dx,1024; Предел стека. Также любой (в примере),

;лишь бы его было достаточно.

mov cl,10010110b; Права доступа дескриптора сегмента

; стека (P = 1, DPL = 00b, S = 1,

; тип = 011b, A = 0).

call set\_descriptor ; Конструируем дескриптор стека.

xor eax,eax; EAX = 0

mov ax,ds

shl eax,4; EAX = физический адрес начала сегмента данных.

mov dx,0ffffh

mov cl,10010010b

call set\_descriptor ; Конструируем дескриптор данных.

mov eax,0b8000h; Физический адрес начала сегмента

; видеопамяти для цветного текстового

; режима 80 символов, 25 строк

mov edx,4000; Размер сегмента видеопамяти (80\*25\*2=4000).

mov cl,10010010b; Права доступа - как сегмент данных

call set\_descriptor; Конструируем дескриптор сегмента

; видеопамяти.

; Готовим дополнительные дескрипторы для возврата в R-Mode:

xor eax,eax

push cs

pop ax

shl eax,4 ; EAX = физический адрес сегмента кода

; (и всех остальных сегментов, т.к.

; это .com-программа)

mov edx,0ffffh

mov cl,10011010b ; P=1, DPL=00b, S=1, Тип=101b, A=0

call set\_descriptor ; R\_Mode\_Code

mov cl,10010010b ; P=1, DPL=00b, S=1, Тип=001b, A=0

call set\_descriptor ; R\_Mode\_Data

; Устанавливаем GDTR:

xor eax,eax; EAX = 0

mov edx,eax; EDX = 0

mov ax,ds

shl eax,4; EAX = физический адрес начала сегмента данных.

lea dx,GDT

add eax,edx; EAX = физический адрес GDT

mov GDT\_adr,eax; Записываем его в поле адреса образа GDTR.

mov dx,55 ; Предел GDT = 8 \* (1 + 6) - 1

mov GDT\_lim,dx; Записываем его в поле предела образа GDTR.

; запрет маскируемых прерываний:

cli ; Запрещаем прерывания. Для того, чтобы прерывания

; работали в защищённом режиме их нужно специально

; определять, здесь это не делается.

; запрет немаскируемых прерываний(которые не запрещаются *cli*):

in AL,70h

or AL,80h

out 70h,AL

lgdt GDTR; Загружаем образ GDTR в сам регистр GDTR.

mov R\_Mode\_SP,sp ; сохраняем указатель на стек

; Переходим в защищённый режим:

mov eax,cr0

or al,1

mov cr0,eax

; Процессор в защищённом режиме

; Этими пятью байтами кодируется

db 0eah ; Команда far jmp Code\_selector:P\_Mode\_entry.

dw P\_Mode\_entry

dw Code\_selector

;----------------------------------------------------

P\_Mode\_entry:

; В CS находится уже не сегментный адрес сегмента кода, а ;селектор его дескриптора.

; Загружаем сегментные регистры. Это обеспечит работу программы

; на любом 32-разрядном процессоре.

mov ax,Screen\_selector

mov es,ax

mov ax,Data\_selector

mov ds,ax

mov ax,Stack\_selector

mov ss,ax

mov sp,0

; выводим ZS-строку о входе в P-Mode:

lea bx,Start\_P\_Mode\_ZS

mov di,480 ; Выводим ZS-строку со смещения 480 в

; видеопамяти (оно соответствует началу

; 3-й строки на экране в текстовом режиме).

call putzs ; вызов функции

; Работа программы в защищённом режиме

; (здесь - только вывод строки):

lea bx,P\_Mode\_ZS

add di,160

call putzs; печатаем

; Возвращаемся в режим реальных адресов. Для этого

; Запретить прерывания (CLI). Они были запрещены при входе в P-;mode.

; Передать управление в читаемый сегмент кода, имеющий предел в ;64Кб,

; сбросить очередь предвыборки и загрузить CS реальным сегментным адресом

db 0eah ;Команда far jmp R\_Mode\_Code:Pre\_R\_Mode\_entry.

dw Pre\_R\_Mode\_entry

dw R\_Mode\_Code

Pre\_R\_Mode\_entry:

;Загрузить в SS,DS,ES,FS и GS селекторы дескрипторов,имеющих

; следующие параметры:

; 1) Предел = 64 Кб (FFFFh)

; 2) Байтная гранулярность (G = 0)

; 3) Расширяется вверх (E = 0)

; 4) Записываемый (W = 1)

; 5) Присутствующий (P = 1)

; 6) Базовый адрес = любое значение

mov ax,R\_Mode\_Data ;Селектор R\_Mode\_Data-"один для всех".

mov ss,ax

mov ds,ax

mov es,ax

mov fs,ax

mov gs,ax

; Сбросить флаг PE в CR0.

mov eax,cr0 ; чтение cr0

and al,0feh ; FEh = 1111'1110b **,** сброс бита 0

mov cr0,eax

;Выполнить far jump на программу режима реальных адресов.

db 0eah

R\_Mode\_offset dw ?; Значения R\_Mode\_offset и R\_Mode\_segment

R\_Mode\_segment dw ?; сюда записаны перед входом в

; защищённый режим (в начале программы).

;------------------------------------------------

R\_Mode\_entry:

; Загрузить в регистры SS, DS, ES, FS и GS необходимые значения

; (восстанавливаем сохранённые значения):

mov ss,R\_Mode\_SS

mov ds,R\_Mode\_DS

mov es,R\_Mode\_ES

mov fs,R\_Mode\_FS

mov gs,R\_Mode\_GS

mov sp,R\_Mode\_SP ; Восстанавливаем указатель стека

; перед разрешением прерываний.

; Разрешить прерывания (STI).

sti

; разрешить NMI

in al,70h

and al,07Fh

out 70h,al

; подождать нажатия любой клавиши

mov ah,0

int 16h

; Выводим ZS-строку "Back to real address mode..."

lea bx,R\_Mode\_ZS

mov ax,0b800h

mov es,ax

mov di,800

call putzs ; Функция putzs универсальна и работает

; в обоих режимах.

int 20h ; Конец программы (выход в MS-DOS).

;--функция-------------------------------------------------

set\_descriptor proc near

; Создаёт дескриптор.

; DS:BX = дескриптор в GDT

; EAX = адрес сегмента

; EDX = предел сегмента

; CL = байт прав доступа (access\_rights)

push eax

push ecx ; Регистры EAX и ECX будем использовать.

push cx ; Временно сохраняем значение access\_rights.

mov cx,ax ; Копируем младшую часть адреса в CX,

shl ecx,16 ; и сдвигаем её в старшую часть ECX.

mov cx,dx ; Копируем младшую часть предела в CX.

; Теперь ECX содержит младшую часть

; дескриптора (т.е. первые 4 байта ).

mov [ bx ],ecx ; Записываем младшую половину

;дескриптора в GDT.

shr eax,16 ; EAX хранит адрес сегмента, младшую часть

; которого мы уже использовали, теперь будем

; работать со старшей, для чего сдвигаем её в

; младшую часть EAX, т.е. в AX.

mov cl,ah; Биты адреса с 24 по 31

shl ecx,24; сдвигаем в старший байт ECX,

mov cl,al; а биты адреса с 16 по 23 - в младший байт.

pop ax ; Возвращаем из стека в AX значение

; access\_rights

mov ch,al; и помещаем его во второй (из четырёх)

; байт ECX.

; дескриптор готов. Старшую часть

; предела и биты GDXU не устанавливаем и

; они будут иметь нулевые значения.

mov [ bx + 4 ],ecx ; Дописываем в GDT вторую половину

; дескриптора.

add bx,8 ; Переводим указатель в GDT для работы

; со следующим дескриптором

pop ecx

pop eax

ret

endp set\_descriptor

;===функция=======================

putzs proc near

; DS:BX = ZS ; ZS = Zero-String - строка, оканчивающаяся

; нулевым (00h) байтом.

; ES:DI = позиция вывода ; ES описывает сегмент видеопамяти,

; DI - смещение в нём.

push ax

push bx

push es

push di

mov ah,1bh ; В AH будет атрибут вывода - светло-циановые

; символы на синем фоне.

putzs\_1:

mov al,[ bx ] ; Читаем байт из ZS-строки.

inc bx ; Переводим указатель на следующий байт.

cmp al,0 ; Если байт равен 0, то получили ZS и

je putzs\_end; переходим в конец процедуры.

mov es:[ di ],ax; Иначе - записываем символ вместе с

; атрибутом в видеопамять по указанному

; смещению - цветной символ появится на

; экране.

add di,2 ; Переводим указатель в видеопамяти на

; позицию следующего символа.

jmp putzs\_1; Повторяем процедуру для следующего байта

; из ZS-строки, цикл

putzs\_end:

pop di

pop es

pop bx

pop ax

ret

endp putzs

;------------------------------------------------------------

; ZS-строка для вывода при входе в P-Mode:

Start\_P\_Mode\_ZS: db "Leaving Real Mode...",0

; ZS-строка для вывода при работе в P-Mode:

P\_Mode\_ZS: db "<Family> working in 32-bit PMode...",0

; ZS-строка для вывода в R-Mode:

R\_Mode\_ZS: db "Back to Real Mode...",0

;-----------------------------------------------------

; Для сохранения регистров до перехода в P-Mode:

R\_Mode\_SP dw ?

R\_Mode\_SS dw ?

R\_Mode\_DS dw ?

R\_Mode\_ES dw ?

R\_Mode\_FS dw ?

R\_Mode\_GS dw ?

;--------------------------------------------------------

; Образ регистра GDTR:

GDTR label fword

GDT\_lim dw ?

GDT\_adr dd ?

;--------------------------------------------------------

GDT:

dd ?,? ; 0-й дескриптор

dd ?,? ; 1-й дескриптор (кода)

dd ?,? ; 2-й дескриптор (стека)

dd ?,? ; 3-й дескриптор (данных)

dd ?,? ; 4-й дескриптор (видеопамяти)

dd ?,? ; 5-й дескриптор (код для перехода в R-Mode)

dd ?,?;6-й дескриптор(стек и данные для перехода в R-Mode)

;-------------------------------------------------------

db 1024 dup (?) ; Зарезервировано для стека.

Stack\_seg\_start:; Последн метка программы-отсюда будет расти

;стек.

;--------------------------------------------------------

main endp

pmode ends

end start

Для компиляция программы использовать tasm, tlink с помощью BAT файла:

* в формате DOS EXE: tasm /m !.asm\\\tlink /3 /x !.obj\\\del !.obj
* в формате DOS COM: tasm /m !.asm\\\tlink /t /3 /x !.obj\\\del !.obj
* Консольная программа для Windows в формате PE EXE: tasm32 /m /mx !.asm\\\tlink32 /Tpe /ap /c /x !.obj /o c:\tasm\lib\import32.lib\\\del !.obj
* Оконная программа для Windows в формате PE EXE: tasm32 /m /mx !.asm\\\tlink32 /Tpe /aa /c /x !.obj /o c:\tasm\lib\import32.lib\\\del !.obj

Для изучения механизма выполнения 32-разрядного сложения надо выполнить программу до команды пересылки содержимого *EAX* в переменную ***sum*** включительно.

Если объявить результирующую переменную как:

sum dd 0 ; Переменная для суммы

то:

;фрагмент программы

mov eax,12345678h ; Первый 32-разрядный операнд

add eax,87654321h ; Второй 32-разрадный операнд

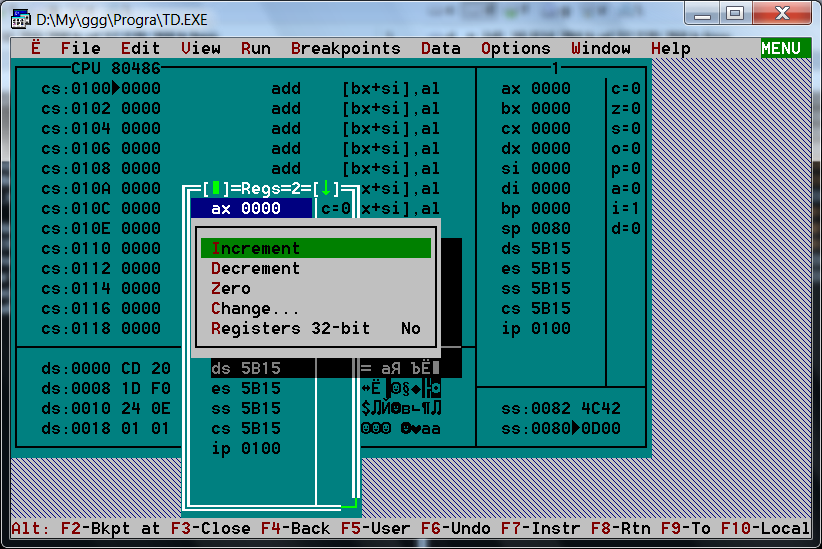
mov dword ptr sum,eax ; Запись результата в sum

Студенты должны вставить данный фрагмент в программу после перехода в защищенный режим. В первом операнде студент указывает дату рождения, во втором –номер студенческого билета.

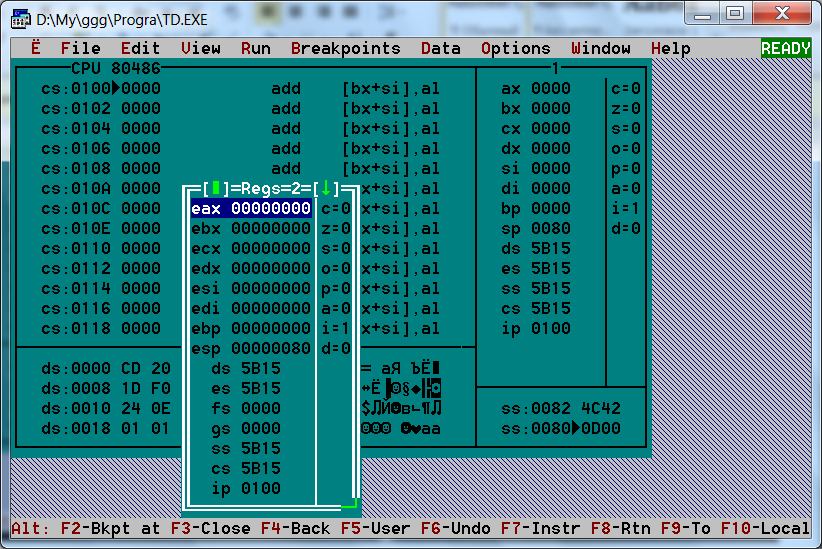
При анализе листинга программы, выделить в отчете команды МП для работы с 32-разрядными операндами.

Для отладки программы используется отладчик (TD.exe) фирмы Borland. Для отображения содержимого 32-разрядных регистров надо после запуска отладчика,

выбрать опцию «*View*→*Registers».* В появившемся окне выбрать *Registers*. Затем необходимо вызвать локальное меню этого окна и нажав *ALT-F10*, выбрать в меню пункт *Registers 32 bit*.



Это обеспечит вывод на экран содержимого полных 32-разрядных регистров *EAX...ESP* взамен 16-разрядных регистров *AX...SP*.



Кроме значения переменной sum в отчет по лабораторной работе из окна просмотра переменных включить ее адрес в виде (ssss:oooo).

При оформлении отчета в таблице GDT найти и указать линейные адреса в памяти базы и лимита для каждого из использованных в программе сегментов.