МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

Системное программирование

Учебно-методическое пособие

Озерск — 2019

УДК 681.3.06
П56

Вл. Пономарев. Системное программирование. Учебно-методическое пособие. Озерск: ОТИ НИЯУ МИФИ, 2019. — 88 с.

В пособии рассматриваются ошибки и исключения, управление памятью, работа с реестром, вопросы безопасности, статические и динамически подключаемые библиотеки, механизмы взаимодействия между процессами (буфер обмена, Data Copy, DDE, почтовые ящики и каналы), сервисы NT.

В качестве основного материала при изучении дисциплины «Системное программирование» пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника», и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1)
2)

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Введение

Системное программирование охватывает программное обеспечение, разработка которого требует детального знания устройства компьютера и операционной системы. Традиционно к области системного программирования относятся базы данных, языковые и инструментальные системы, драйверы и т.п., требующие от программиста высокой квалификации, а от программы — высокой степени отлаженности, надежности и безопасности.

В настоящем пособии рассматриваются следующие темы.

1. Символьные и строковые типы данных Windows.
2. Обработка ошибок, вывод сообщений об ошибках.
3. Структурированная обработка исключений Windows.
4. Реестр Windows.
5. Структуры безопасности Windows.
6. Управление памятью.
7. Статические и динамические библиотеки.
8. Межпроцессное взаимодействие.
9. Сервисы NT.

Текст данного пособия время от времени совершенствуется, поэтому рекомендуется использовать его электронную версию, размещенную на сайте автора revol.ponocom.ru.

Изучение материала сопровождается выполнением практических работ, описанных в учебно-методическом пособии «Практикум по системному программированию», электронная версия которого размещена на сайте автора.

1. Символы и строки Windows

1.1. Символы

Для представления знаковых (символьных) и строковых данных в системе Windows используется множество типов, макросов и функций. Типами для знаковых данных являются узкие и широкие (wide) символы. Узкий символ занимает в памяти один байт, широкий — два байта.

Узкий символ — это стандартный тип `char`. Широкий символ определяется в файле ctype.h или basetyps.h следующим образом:

```
typedef unsigned short wchar_t;
```

то есть как беззнаковое 16-битное целое. При этом неявно подразумевается, что тип `wchar_t` предназначен для кодировки Unicode, в которой для представления одного знака как раз используется 16 бит.

Основной причиной возникновения множества типов для представления строк является наличие множества кодировок для представления самих знаков. Так, первоначально подавляющее большинство платформ (операционных сред) использовали для представления знаков таблицы кодировок ASCII, что почти равнозначно таблицам кодировок ANSI. Для представления одного символа в этих таблицах используется один байт. Язык программирования Си внес некоторую путаницу в представление знаков, определив знак как 8-битное целое число со знаком. Это означает, что символы из второй половины таблицы кодировки формально имеют отрицательные коды ASCII.

Одновременно для представления отдельных символов была разработана система Unicode, в которой символ кодируется 16-битным целым числом без знака. Цель Unicode — обеспечить кодировку всех возможных символов, включая китайский, японский и корейский алфавиты, насчитывающие до 5 тысяч знаков.

Кодировка Unicode, однако, не поддерживалась большинством платформ. Для кодировки символов японского, китайского и корейского алфавитов были предложены различные способы записи, основанные на однобайтном представлении (на кодировке ANSI). Эти способы получили название DBCS (double-byte character set — двухбайтный набор символов) и MBCS (multi-byte character set — многобайтный набор символов). В принципе, DBCS и MBCS — это одно и то же. Проблема с подобными наборами заключается в том, что некоторые символы в них являются лидирующими, то есть используются для идентификации одного или нескольких последующих байт. Эти проблемы, однако, нас не касаются, и в дальнейшем мы не будем рассматривать наборы типа DBCS. Они просто создают еще большую путаницу.

1.2. Платформы

Платформа — это специфичное операционное окружение, внутри которого разрабатываются и функционируют программы. Если говорить о платформах Windows, то существуют платформы Win16, Win32 и WinNT (и это не все), которые определяют установки по умолчанию, форматы файлов, средства построения результирующих файлов и т.п.

Так, платформа Win16, на которой построена операционная система Windows 3.x, определяет, что строки — это массивы символов ANSI, и символы Unicode для этой платформы являются неизвестным понятием. Системные функции этой платформы не предназначены для работы со строками Unicode. Платформа Win32 поддерживает Unicode в каком-то неполном объеме, и является переходной к платформе WinNT, в которой поддержка Unicode полная. Что означает «полная» и «неполная» поддержка, точно неизвестно. Замечу только, что файловые системы Windows, начиная с версии 98-2, поддерживают Unicode для именования файлов и каталогов (папок).

Обеспечение межплатформенной совместимости является одной из важнейших черт средств программирования. Для этой цели в Windows введен еще один тип символов, который является совместимым с любой платформой — тип _TCHAR (TCHAR). Во время компиляции программы он определяется как однобайтный или двухбайтный символ в зависимости от платформы, для которой строится программа. Для этого типа введено понятие символа общего назначения — Generic char.

В рамках проекта Microsoft Visual Studio при программировании на Си или C++ указание платформы не определяет знаковый тип. Этот тип должен быть указан в настройках проекта как многобайтный или как широкий. В зависимости от этого тип TCHAR представляет собой узкий или широкий символ, а макрос, например, CreateWindow, расширяется в функцию CreateWindowA, или в функцию CreateWindowW. При этом программист может использовать эти функции явно, игнорируя настройки проекта и определенные этими настройками символы компилятора, управляющие трансляцией типа TCHAR.

Для различения платформ Win16 и Win32 введены типы: CHAR — это char для Win32, а WCHAR — это wchar_t для Win32. Возникновение технологий OLE и COM также внесло свою лепту в путаницу с символами. Изначально технология COM предполагала использование только кодировок Unicode. Для обеспечения совместимости с OLE в Windows определен тип OLECHAR, который, понятно, есть wchar_t.

Появление платформы Microsoft .NET несколько улучшило ситуацию в том смысле, что в рамках платформы символьные и строковые типы данных основаны на символах Unicode.

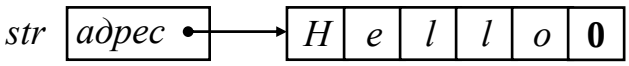
1.3. Строки

Строка — это упорядоченное множество знаков, определяемое указателем. Строкового типа в машинном представлении нет, и это создает определенные проблемы, если язык программирования не определяет строкового типа.

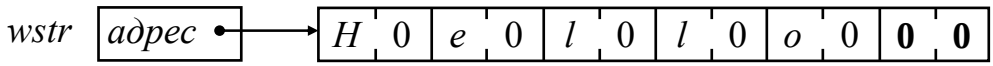
Во многих языках строковые типы определены. Для примера можно привести Pascal, Visual Basic, Java, C# и другие. При программировании на Си или C++ можно воспользоваться классом CString библиотеки MFC, или классом string, определенным в пространстве имен std. Если этого недостаточно, есть еще тип BSTR, представляющий собой строку широких символов размером до почти 4 Гбайт.

Все эти встроенные или внешние механизмы управления строками хорошо работают при программировании в рамках одной среды и проекта, но все становится хуже в системном программировании, когда требуется слияние технологий, инструментов, языков, модулей и т.п.

В системном программировании наиболее часто используется строковый тип, называемый null-terminated string, и представляющий собой массив знаков с завершающим нулем. Последним элементом такого массива является знак с кодом 0, не являющийся никаким печатаемым символом:



Такое представление строки получило распространение в системном программировании в связи с языком Си. Заметим, что представление строки широких символов принципиально ничем не отличается от представления узких символов, разве что завершающих нуля два:



На практике почти любая строка является либо той, либо другой, поскольку есть лишь два типа символов. Однако развитие разных систем программирования, технологий и платформ привело к появлению большого числа определений строковых типов, построенных на основе двух выше упомянутых представлений в виде массивов.

Кроме проблем с выделением и освобождением памяти, представление строк в виде массивов привносит проблему приведения одних типов данных к другим, а также преобразования одних строк в другие.

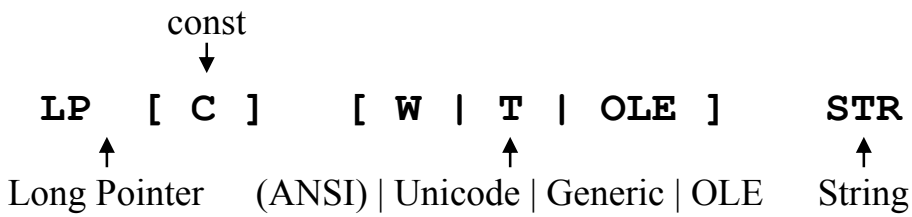
Если в стандартном Си единственными функциями сравнения строк были strcmp и strcmpi, то множество символьных типов в Windows породило соответствующее множество строковых, то есть указательных типов и, соответственно, множество функций для работы с ними.

Стремление поддерживать старые форматы данных, библиотеки и системы программирования привело к тому, что в одной системной программе одновременно используются сразу несколько строковых типов. Мало того, что они могут оказаться несовместимыми по объему занимаемой памяти (узкими или широкими), необходимо еще обеспечить совместимость на уровне компилятора, на момент контроля типов. Две строки, которые кажутся программисту совершенно одинаковыми и фактически совершенно одинаково размещены в памяти, компилятор считает несовместимыми, если они имеют разные указательные типы. В большинстве случаев проблема несовместимости в этом случае решается за счет явного приведения типа, как в следующем примере кода:

```
#include <windows.h>
#include <tchar.h>
int main() {
    char * str_a = 0;
    LPCTSTR str_t = _T("abc");
    str_a = (char*)str_t;
    return 0;
}
```

Здесь используется макрос _T, формирующий литерал символов общего типа. В зависимости от настройки кодировки в свойствах проекта, выполнение этого кода приводит к разным результатам. Заметим, что существует идентичный макрос _Text. Как тут не запутаться.

Для определения строковых типов в Windows используются определения типов typedef, начинающиеся с LP — Long Pointer. Следующий рисунок поясняет, как определить тип указателя:



Так, например, LPCSTR — это постоянная строка ANSI, const char*, а LPOLESTR — это указатель на строку OLE, OLECHAR*, или wchar_t*. Стандартные системные функции Windows используют обычно один из таких указательных типов.

Для перевода символов строк из широких в узкие и обратно используются макросы, доступные после объявления USES_CONVERSION.

```
A2CW    (LPCSTR)    -> (LPCWSTR)
A2W     (LPCSTR)    -> (LPWSTR)
W2CA    (LPCWSTR)   -> (LPCSTR)
W2A     (LPCWSTR)   -> (LPSTR)
```

Для перевода строк OLE используются следующие макросы:

```
T2COLE   (LPCTSTR)    -> (LPCOLESTR)
T2OLE    (LPCTSTR)    -> (LPOLESTR)
OLE2CT   (LPCOLESTR)  -> (LPCTSTR)
OLE2T    (LPCOLESTR)  -> (LPCSTR)
```

При необходимости эти макросы выполняют преобразование типов при помощи системных функций, для этой цели предназначенных, а именно, — при помощи основных функций MultiByteToWideChar и WideCharToMultiByte. Ничто не мешает использовать эти функции прямо, если помнить, какие параметры (до восьми) требуются. MultiByte в данных функциях означает однобайтные наборы (символы ANSI).

Для обычных операций со строками теперь следует использовать модернизированные версии строковых функций strcpy, strcmp, strlen и других. Их так много, что одно только перечисление займет целый лист.

Еще одна проблема со строками в OLE — определение строковых констант символами Unicode:

```
OLECHAR* pOLEStr;
pOLEStr = L"Hello";
```

Однако такой прием может привести к проблемам на платформах, не поддерживающих Unicode. Поэтому существует макрос OLESTR, который определяется по-разному в зависимости от платформы:

```
pOLEStr = OLESTR("Hello")
```

Напоследок приведем таблицу основных типов (таблица 1).

Таблица 1. Основные символьные и строковые типы Windows

Тип	Описание
char	8-битный знаковый символ (символ ANSI)
wchar_t	16-битное целое без знака (символ Unicode)
CHAR	Версия char для Win32
WCHAR	Версия wchar_t для Win32
OLECHAR	Версия wchar_t для OLE
_TCHAR	Символ общего типа — char или wchar_t
LPSTR, LPCSTR	Указатель на символ для Win32
LPWSTR, LPCWSTR	Указатель на широкий символ для Win32
LPOLESTR, LPCOLESTR	Указатель на широкий символ для OLE
LPTSTR, LPCTSTR	Указатель на символ общего типа для Win32
_T(str), _TEXT(str)	Макросы для формирования строк общего типа
OLESTR(str)	Макрос для формирования строки общего типа

2. Обработка ошибок

В основе системного программирования для операционной системы Windows лежит использование функций API (системных функций). Эти функции во многих случаях выполняются успешно, однако некоторые функции могут завершаться неудачно. В задачу программиста входит отслеживание возникающих ошибок и выработка стратегии по их локализации и обработке.

2.1. Код последней ошибки

С каждым потоком Windows сопоставляется код последней ошибки. Это 32-битное целое число без знака, хранящее код ошибки, вызванной выполнением той или иной системной функции.

Описание структуры кода приводится в начале файла winerror.h:

```
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +---+-+-+-----------------------+-------------------------------
//  |Sev|C|R|     Facility          |               Code
//  +---+-+-+-----------------------+-------------------------------
//  where
//      Sev - is the severity code
//          00 - Success
//          01 - Informational
//          10 - Warning
//          11 - Error
//      C - is the Customer code flag
//      R - is a reserved bit
//      Facility - is the facility code
//      Code - is the facility's status code
```

Два старших бита кода отвечают за серьезность ошибки. Значение этих битов, равное 11, говорит о том, что произошла ошибка, на которую нужно реагировать. Младшие 16 бит содержат код, описывающий ошибку, а биты 16-27 указывают на подсистему (facility).

Далее в файле winerror.h описывается более десяти тысяч ошибок, которые могут возникать (начало описания ошибок):

```
// MessageId: ERROR_SUCCESS
// The operation completed successfully.
#define ERROR_SUCCESS                    0L
#define NO_ERROR                         0L
#define SEC_E_OK                         ((HRESULT)0x00000000L)
//
// MessageId: ERROR_INVALID_FUNCTION
// Incorrect function.
#define ERROR_INVALID_FUNCTION           1L
//
// MessageId: ERROR_FILE_NOT_FOUND
// The system cannot find the file specified.
#define ERROR_FILE_NOT_FOUND             2L
```

С каждой ошибкой связан код, константа и краткое описание.

Константы используются программистами, так как они понятнее кодов. Описания ошибок хранятся в DLL, специально для этой цели предназначенных. Эти описания могут быть сформированы на разных языках. При установке операционной системы в нее записываются DLL с описаниями ошибок на том языке, который был выбран как язык локализации операционной системы.

Некоторые программные компоненты могут генерировать свой собственный набор ошибок, предоставляя свои собственные DLL с описаниями. Некоторые из этих компонентов, например, OLE, нашли отражение в файле winerror.h, некоторые компоненты генерируют ошибки, коды которых отсутствуют в файле winerror.h.

2.2. Функции для работы с кодом ошибки

Код последней ошибки можно получить и установить. Возвращает код функция GetLastError. Параметров у функции нет, возвращаемое значение имеет тип DWORD. Эта функция вызывается при необходимости после вызова системной функции, которая предположительно может выполниться неуспешно.

Устанавливает код последней ошибки функция SetLastError, единственный параметр которой — код ошибки, который требуется установить. Обычно устанавливается нулевое значение, свидетельствующее об отсутствии ошибки. Однако установить можно любое значение, например, в целях отладки, для имитации некоторой критической ситуации. При этом следует использовать константу, описывающую ошибку. Например, следующий код моделирует ситуацию отсутствия файла:

```
SetLastError(ERROR_FILE_NOT_FOUND);
```

Следующие примеры, напротив, очищают код ошибки:

```
SetLastError(ERROR_SUCCESS);
SetLastError(NO_ERROR);
SetLastError(0);
```

Функция SetLastError вызывается до вызова системной функции.

По замыслу разработчиков, каждая системная функция устанавливает код ошибки. Это дает возможность программисту отслеживать ход выполнения системных функций и в случае необходимости корректировать алгоритм программы. Однако на самом деле не все функции поступают так. Типичный пример: функция завершилась успешно, и должна была бы установить нулевой код. Так как часть функций в случае успеха ничего не устанавливают, код остается неизменным. Тогда, если ранее код был установлен, можно подумать, что ошибка вызвана текущим вызовом.

Поэтому, если есть необходимость контролировать код ошибки, перед вызовом системной функции устанавливается нулевой код, а после вызова код проверяется. Если код изменился, ошибка произошла.

Примерная структура кода при этом следующая:

```
SetLastError(NO_ERROR);
Вызов-системной-функции
DWORD dwError = GetLastError();
```

На практике код последней ошибки обычно проверяется тогда, когда системная функция возвращает признак неудачи. Для этого нужно точно знать, что возвращает системная функция.

Структура кода при этом может иметь следующий вид:

```
BOOL result = Вызов-системной-функции
if (!result) {
     DWORD dwError = GetLastError();
}
```

Полезно отслеживать код ошибки в процессе отладки программы. Для этого в окно Watch нужно ввести значение «@err, hr» (рисунок 1).

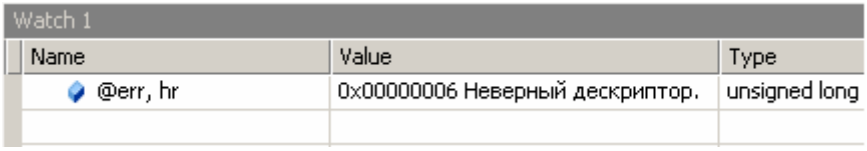


Рисунок 1 — Отображение последней ошибки в окне Watch

С Visual Studio 6.0 поставляется также утилита Error Lookup, которая позволяет получать описание ошибки по ее коду (рисунок 2).

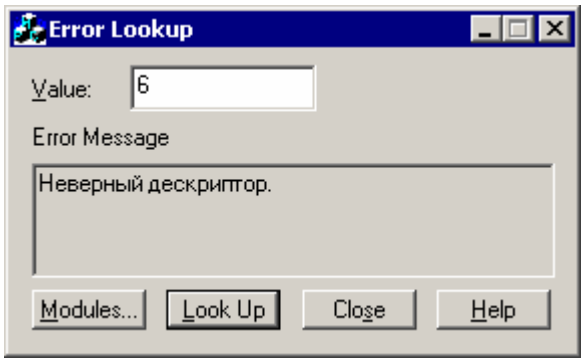


Рисунок 2 — Утилита Error Lookup

Эта утилита может самостоятельно выбирать код ошибки из среды разработки Microsoft Visual Studio, если код предварительно выделить.

Эта утилита позволяет также выбрать модуль, который содержит описания ошибок, если этот модуль вам известен.

2.3. Что возвращают системные функции

Системные функции могут возвращать разные результаты. Иногда возвращаемый результат является признаком успешности выполнения функции, иногда служит дополнительным признаком состояния выполнения задачи. В таблице 2 показаны типы данных возвращаемых значений большинства функций API Windows.

Таблица 2. Стандартные типы значений, возвращаемых функциями API

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно.
BOOL	В случае неудачи возвращается 0. В остальных случаях возвращаемое значение не 0.
HANDLE	В случае неудачи обычно возвращается NULL. В остальных случаях HANDLE идентифицирует объект. Иногда возвращается значение INVALID_HANDLE_VALUE, равное –1, свидетельствующее о неудаче.
PVOID	В случае неудачи возвращается NULL. В случае успеха PVOID сообщает адрес блока данных.
LONG или DWORD	Функции, которые возвращают значения счетчиков. В случае неудачи эти функции возвращают 0 или –1.

Некоторые системные функции всегда завершаются успешно, но по разным причинам. Например, попытка создать поименованный объект ядра «событие» или «мьютекс» может быть успешна либо потому, что объект действительно создан, либо потому, что такой объект уже есть.

Иногда нужно знать причину успеха. Для этого Microsoft использует механизм установки кода последней ошибки. При успешном выполнении некоторых функций вызов GetLastError дает дополнительную информацию. Например, в случае, если поименованный объект типа «событие» или «мьютекс» уже существует, функция GetLastError возвращает значение константы ERROR_ALREADY_EXISTS. В этом случае код создания мьютекса может иметь следующую структуру:

```
HANDLE hMutex = CreateMutex(0, 0, "mutex-name");
if (hMutex == NULL) {
     // мьютекс не создан
     // получим код ошибки
     DWORD dwError = GetLastError();
} else if (GetLastError() == ERROR_ALREADY_EXISTS) {
     // мьютекст существует
} else {
     // мьютекст создан
}
```

Особый случай возникает также при создании объекта «файл» при помощи системной функции CreateFile. В случае неудачи эта функция возвращает INVALID_HANDLE_VALUE.

Создание файла проверяется в этом случае следующим образом:

```
HANDLE hFile= CreateFile( . . . );
if (hFile == INVALID_HANDLE_VALUE) {
     // ошибка создания
     DWORD dwError = GetLastError();
}
```

2.4. Реакция на ошибку

Если программа обнаруживает ошибку, она должна предпринять некоторые действия. Возможны разные варианты реакции программы на возникающие ошибки.

2.4.1. Отсутствие контроля ошибок

Программа не контролирует ошибки и ничего не предпринимает в надежде, что ошибка не окажет влияния на общий результат работы программы. Несмотря на кажущуюся нелепость, этот вариант может использоваться во многих случаях. Более того, часто программисты пишут программы, не контролируя возникающие ошибки, иначе говоря, они выбирают как раз этот вариант. Обычно не возникает ошибок в функциях, возвращающих логические значения, и предназначенных для получения некоторой информации. Например, функция GetClientRect, возвращающая размер клиентской части окна, редко завершается неудачно. Можно не контролировать ошибки, возникающие при установке параметров интерфейса. Программа может установить тип сглаживания шрифтов или изменить системный цвет. Если при этом возникнет ошибка, на работу программы это не окажет существенного влияния.

2.4.2. Альтернативные варианты алгоритма

Программа изменяет ход выполнения алгоритма программы с тем, чтобы возникшая ошибка не повлияла на ее общий результат.

В качестве примера можно рассмотреть программу, использующую слоеные окна. Если при создании окна произвольной конфигурации функция SetLayeredWindowAttributes завершилась неудачно, программа может создать обычное прямоугольное окно.

Таких случаев не так много. Обычно, если системная функция завершилась с ошибкой, требуется серьезная реакция. Это относится в первую очередь к функциям, создающим объекты ядра, такие, как процессы, потоки, файлы и т.п. В этих случаях следует использовать третий вариант.

2.4.3. Взаимодействие с пользователем

При обнаружении ошибки, ведущей к невозможности выполнения алгоритма программы, программа уведомляет пользователя при помощи окна сообщения. В этом случае возможны два сценария развития событий. Первый сценарий сообщает пользователю об ошибке и завершает работу программы. Этот сценарий является одним из наиболее распространенных.

Второй сценарий предусматривает взаимодействие с пользователем с целью получить дополнительные указания. Типичный пример — попытка открыть несуществующий файл. В этом случае программа может запросить у пользователя спецификацию другого файла. Другой типичный пример — предложить варианты развития событий и спросить пользователя, какой вариант выбирает он.

2.5. Системные сообщения об ошибках

Если приложение обнаруживает ошибку, оно может уведомлять пользователя при помощи функции MessageBox. Для получения описания ошибки следует использовать функцию FormatMessage, которая преобразует код ошибки в описание на языке пользователя.

```
DWORD FormatMessage (
    DWORD dwFlags,          // флаги
    LPCVOID pSource,        // источник сообщения
    DWORD dwMessageId,      // код ошибки
    DWORD dwLanguageId,     // идентификатор языка
    PTSTR pszBuffer,        // буфер для сообщения
    DWORD nSize,                 // размер буфера
    va_list *Arguments      // аргументы сообщения
);
```

Пример применения этой функции:

```
HLOCAL hlocal = NULL;   // буфер для сообщения
DWORD dwError = GetLastError();
BOOL fResult = FormatMessage(
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
    NULL,
    dwError,
    MAKELANGID(LANG_RUSSIAN, SUBLANG_DEFAULT),
    (LPTSTR) & hlocal,
    0,
    NULL);
if (hlocal || !fResult) {
    MessageBox( 0, (LPCTSTR)hlocal, 0, MB_OK );
    LocalFree( hlocal );
} else {
    MessageBox( 0, "Error FormatMessage.", 0, MB_OK );
}
```

В данном примере кода формируется системное сообщение, при этом буфер для сообщения резервирует система (на это указывают флаги). Поскольку сообщение формируется системой, источник описания равен нулю. В других случаях он может указывать на программный модуль, например, на DLL.

При помощи макроса MAKELANGID формируется идентификатор языка, который необходимо использовать для формирования сообщения. Поскольку буфер резервирует система, вместо размера буфера подставляется нулевое значение.

Параметры используются тогда, когда нужно подставлять фактические значения в сообщение, например, номер или имя диска и т.п.

2.6. Функция MessageBox

Функции MessageBox не существует, это макрос. При компиляции он расширяется в функцию MessageBoxA, или в функцию MessageBoxW. Первая функция используется строки ANSI, вторая — строки Unicode.

Эта функция имеет много практических применений.

Рассмотрим параметры функции.

1. Дескриптор родительского окна. Окно сообщения центрируется относительно родительского окна. Если родительского окна нет (например, программа консольная), вместо дескриптора подставляется NULL, в этом случае рабочий стол будет являться родительским окном.

2. Текст сообщения. Это либо непосредственный литерал, либо буфер, содержащий текст сообщения. Буфер обычно используется тогда, когда нужно сформировать текст с числовыми значениями.

3. Заголовок окна сообщения. Если этот параметр равен NULL, заголовок будет «Error».

4. Стиль окна сообщения. Именно этот параметр задает различные применения функции. Параметр есть сумма нескольких констант.

Следующие константы задают значок в окне.

MB_ICONEXCLAMATION — значок предупреждающего сообщения.

MB_ICONINFORMATION — значок информационного сообщения.

MB_ICONQUESTION — значок вопросительного сообщения.

MB_ICONSTOP — значок критического сообщения.

Следующие константы задают кнопки и надписи на них.

MB_OK — только кнопка Ok.

MB_OKCANCEL — кнопки Ok и Отмена.

MB_YESNO — кнопки Да и Нет.

MB_YESNOCANCEL — кнопки Да, Нет и Отмена.

MB_RETRYCANCEL — кнопки Повторить и Отмена.

MB_ABORTRETRYIGNORE — Прервать, Повторить, Игнорировать.

Следующие константы задают кнопку по умолчанию. Эту кнопку пользователь сможет нажать клавишей Enter. Заданы 4 константы, различающиеся цифрой в конце. Если кнопкой по умолчанию должна быть первая из кнопок окна, используется константа MB_DEFBUTTON1. К выбору кнопки по умолчанию следует подходить ответственно. Если сообщение используется для получения ответа на вопрос «Удалить этот файл?» и в окне две кнопки, Да и Нет, то кнопкой по умолчанию должна быть вторая из кнопок.

Следующие константы задают модальность окна сообщения.

MB_APPLMODAL — окно модально относительно приложения.

MB_SYSTEMMODAL — окно модально относительно системы.

Окно, модельное относительно системы, будет появляться поверх всех других окон. Его пользователь точно увидит. Окно, модальное относительно приложения, будет появляться поверх всех окон приложения. Если приложение не активно, пользователь его может и не увидеть.

Положение окна задают также следующие константы.

MB_SETFOREGROUND — окно выводится на передний план.

MB_TOPMOST — окно становится самым верхним.

MB_SERVICE_NOTIFICATION — окно выводится даже тогда, когда не выполнен вход в систему. Этот стиль используют сервисы системы.

Программа узнает нажатую пользователем кнопку, анализируя возвращаемое значение, которое может быть одним из следующих.

IDABORT — нажата кнопка Прервать.

IDCANCEL — нажата кнопка Отмена.

IDIGNORE — нажата кнопка Игнорировать.

IDNO — нажата кнопка Нет.

IDOK — нажата кнопка Ok.

IDRETRY — нажата кнопка Повторить.

IDYES — нажата кнопка Да.

Следующий пример кода показывает, как сформировать вопрос и получить ответ пользователя:

```
int result = MessageBox(NULL, "Удалить файл?", NULL,
    MB_YESNO | MB_ICONQUESTION | MB_DEFBUTTON2);
switch (result) {
case IDYES:
    // выбрана кнопка Да
    break;
case IDNO:
    // выбрана кнопка Нет
    break;
}
```

2.7. Обработка ошибок на месте

Ошибочные ситуации в программах не редкость. Примером такой ситуации является попытка открыть несуществующий файл.

Обычно для этой цели программист описывает функцию, которая возвращает логическое значение Истина в случае успеха, и значение Ложь в случае неудачи. Часть кода, которая проверяет возвращаемое значение, осуществляет так называемую обработку ошибки на месте:

```
int result = open_file( . . . );
if (result == 0) {
    // обнаружена ошибка
    // обработка ошибки на месте
}
```

Обработка ошибки на месте является самым распространенным способом обнаружения и обработки ошибочных ситуаций в программе. Характерным примером является передача результата выполнения части кода на ассемблере при помощи установки флага переполнения C:

```
        call    some_proc
        jnc     no_of_error
        ; флаг C установлен
        ; обработка ошибки на месте
        jmp     final
no_of_error:
        ; успешное завершение
final:
        ; окончание обработки
```

Подобная практика привела к появлению механизмов обработки ошибок, основанных на значении специальной переменной, хранящей код последней ошибки. Функция GetLastError, например, возвращает такой код. Другим примером служит специальная переменная errno, хранящая код ошибки при выполнении файловой операции:

```
FILE * file = fopen("a.txt", "rt");
if (file == NULL) {
    if (errno == ENOENT) {
        // не найден файл или путь
    } else if (errno == EPERM) {
        // операция не разрешена
    } else if (errno == EACCESS) {
        // отказано в доступе
    } else if (errno == ENOMEM) {
        // не хватает памяти
    }
}
```

Обработка ошибок на месте затеняет основной код программы, и затрудняет ее прочтение и понимание алгоритма. Тем не менее, этот механизм используется чаше всего.

3. Обработка исключений

Исключительной называется ситуация, в которой дальнейшее выполнение программы невозможно без нарушения ее правильной работы. Примерами таких ситуаций являются попытка деления на ноль, переполнение арифметической операции, переполнение стека, нарушение доступа к памяти. Эти ситуации генерируют исключение (exception).

Исключение завершает выполнение части кода, в которой ситуация возникла, и вызывают переход управления к той части кода, в которой эту ситуацию предполагается исправить. Эту часть кода называют обработчиком исключения (exception handler). Если исключение возникло, и в программе нет обработчика, управление будет передано системному обработчику, который выведет сообщение об ошибке и завершит работу программы.

Различают программные и аппаратные исключения. Программные исключения генерирует программа (программист) при помощи специальной функции. Аппаратные исключения возникают в аппаратуре.

3.1. Структурированная обработка исключений

Структурированная обработка исключений (structured exception handling, SEH) — это механизм операционной системы Windows, предназначенный для обнаружения и обработки исключительных состояний.

Заметим, что этот механизм обработки исключений не является частью какого-либо языка программирования. Языки программирования могут формировать свои собственные механизмы, основанные, или не основанные на механизме SEH. Примером являются исключения C++, изучаемые в курсе объектно-ориентированного программирования.

Общий принцип использования механизма обработки исключений заключается в том, что часть кода, предположительно ведущую к исключительной ситуации, заключают в блок try {}. В случае, если выполнение кода блока try {} привело к исключению, выполнение этого блока немедленно завершается и управление передается обработчику.

Структурированная обработка исключений предусматривает два варианта использования блока try.

Первый вариант предназначен для обработки исключений. В этом случае используются блоки try и except:

```
__try {
    // потенциально опасный код
} __except(filter) {
    // обработка
}
```

При возникновении исключения в блоке try управление переходит в блок except, и дальнейшие действия определяются фильтром, обозначенным в примере как filter. Если исключения не возникло, блок except игнорируется.

Второй вариант предназначен для выполнения очистки памяти, освобождения ресурсов, закрытия файлов и других подобных действий, которые нужно выполнить в случае возникновения исключения. В этом случае используются блоки try и finally:

```
__try {
    // потенциально опасный код
} __finally {
    // очистка
}
```

В этом варианте блок finally выполняется в любом случае, независимо от того, возникло или нет исключение в блоке try.

3.2. Использование блока завершения

Использование блока завершения полезно рассмотреть на примере входа в критическую секцию. Следующий пример кода показывает, как выйти из критической секции в любом случае:

```
DWORD ThreadFunc(PVOID) {
    . . .
    __try {
        // вход
        WaitForSingleObject(hMutex, INFINITE);
        // критическая секция
        return 0;
    } __finally {
        // выход
        ReleaseMutex(hMutex);
        return 1;
    }
}
```

Здесь, независимо от того, что произойдет в критической секции, она будет освобождена в блоке finally.

Следует обратить внимание на операторы return. Выполнение первого из них не завершает выполнение функции. Вместо этого значение 0 будет записано в локальную переменную. После этого выполняется код блока завершения, после чего выполняется второй оператор возврата, записывая в ту же переменную значение 1, которое возвращается как результат функции.

Попытка обойти блок finally при помощи, например, оператора goto приведет к тому, что сначала будет выполнен блок завершения, а затем переход оператора goto. Рекомендуемый порядок формирования кода завершения заключается в том, чтобы использовать вспомогательную переменную (в примере переменная result):

```
DWORD ThreadFunc(PVOID) {
    DWORD result = 0;
    . . .
    __try {
        // вход
        WaitForSingleObject(hMutex, INFINITE);
        // критическая секция
        result = 0;
    } __finally {
        // выход
        ReleaseMutex(hMutex);
        result = 1;
    }
    return result;
}
```

3.3. Использование обработчика

В случае необходимости перехватить исключение и компенсировать ошибочную ситуацию, следует использовать обработчик. При этом блок except получает в качестве параметра арифметическое выражение, результат которого может быть одним из следующих:

- EXCEPTION_CONTINUE_EXECUTION (-1) — исключение игнорируется, продолжается выполнение блока try с оператора, вызвавшего исключение;
- EXCEPTION_CONTINUE_SEARCH (0) — исключение не распознано, управление передается вышестоящему обработчику;
- EXCEPTION_EXECUTE_HANDLER (1) — исключение распознано, выполняется код обработчика, следующий за except.

Никакие другие значения в блок except не должны поступать. Эти значения передаются одним из следующих способов:

- константой в явном виде;
- в виде выражения с использованием логических операций;
- в виде функции.

В любом случае выражение блока except называют фильтром.

3.3.1. Использование явного значения

В этом случае параметром except является константа в явном виде. Она указывает действия, которые должны быть выполнены при возникновения исключения.

Следующий пример показывает, как перехватить исключение при попытке делении на ноль:

```
int main() {
    int x = 1, y = 0;
    __try {
        // деление на ноль
        x = x / y;
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        // задаем значение
        x = 0;
    }
    return 0;
}
```

Значение EXCEPTION_EXECUTE_HANDLER, передаваемое блоку except, указывает, что нужно выполнить код обработчика, который описан в блоке. В данном случае мы задаем какое-то значение переменной x.

Кроме константы EXCEPTION_EXECUTE_HANDLER, можно использовать константу EXCEPTION_CONTINUE_SEARCH. Она вызывает передачу исключения вышестоящему обработчику.

Сначала рассмотрим простой вариант:

```
int main() {
    __try {
        y = 5 / x;
    } __except(EXCEPTION_CONTINUE_SEARCH) {
        printf("Exception raised\n");
    }
    return 0;
}
```

Выполнение этого фрагмента кода имеет такой же эффект, какой возникает при отсутствии обработчика — системный обработчик выводит сообщение и завершает программу. Чтобы продемонстрировать этот вариант, понадобится дополнительная функция и два обработчика:

```
void foo() {
    __try {
        y = 5 / x;
    } __except(EXCEPTION_CONTINUE_SEARCH) {
        printf("Exception raised in foo\n");
    }
}
int main() {
    __try {
        foo();
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        printf("Exception raised in main\n");
    }
    return 0;
}
```

Исключение выбрасывается в функции foo, которая вызывается в функции main. В блоке except обработчика функции foo указана константа EXCEPTION_CONTINUE_SEARCH, поэтому исключение передается обработчику, установленному ранее в функции main. В этом обработчике используется константа EXCEPTION_EXECUTE_HANDLER, предписывающая выполнить код except, и вывести сообщение.

Вследствие того, что в блоке except обработчика функции foo не выполняет никаких действий, блок try-except этой функции бесполезен, и его можно просто удалить. Следующий пример кода является эквивалентным предыдущему, но создающим более простой машинный код:

```
void foo() {
    y = 5 / x;
}

int main() {
    __try {
        foo();
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        printf("Exception raised in main\n");
    }
    return 0;
}
```

Использовать константу EXCEPTION_CONTINUE_EXECUTION непосредственно в фильтре блока except нельзя, так как она заставляет заново выполнить код, вызвавший исключение. Причем повторно выполняется не инструкция языка высокого уровня, а машинная инструкция, которая вызвала исключение. В связи с этим практически очень сложно построить обработчик, который бы исправлял что-то в блоке try с тем, чтобы повторное выполнение не вызывало исключения.

Из всего этого следует, что в качестве фильтра блока except можно использовать только константу EXCEPTION_EXECUTE_HANDLER.

3.3.2. Использование выражения

Выражение используется чаще, чем непосредственное значение в виде константы. Для формирования выражения используется функция GetExceptionCode, которая возвращает код ошибки, вызвавшей исключение. Выражение должно быть составлено таким образом, чтобы в любом случае формировалось значение 0 (или EXCEPTION_CONTINUE_SEARCH) или 1 (или EXCEPTION_EXECUTE_HANDLER).

Для аппаратных исключений определены константы кодов ошибок, приведенные в таблице 3.

Таблица 3. Некоторые константы аппаратных исключений

Константа	Исключительная ситуация
STATUS_ACCESS_VIOLATION	нарушение доступа к памяти
STATUS_FLOAT_DIVIDE_BY_ZERO	вещественное деление на ноль
STATUS_FLOAT_OVERFLOW	вещественное переполнение сверху
STATUS_FLOAT_UNDERFLOW	вещественное переполнение снизу
STATUS_ILLEGAL_INSTRUCTION	недопустимая инструкция
STATUS_INTEGER_DIVIDE_BY_ZERO	целочисленное деление на ноль
STATUS_INTEGER_OVERFLOW	целочисленное переполнение
STATUS_PRIVILEGED_INSTRUCTION	привилегированная инструкция

Часть констант в этом списке опущена, например, константы исключений, используемых отладчиками.

Пример выражения, фильтрующего исключение деления на ноль:

```
(GetExceptionCode() == STATUS_INTEGER_DIVIDE_BY_ZERO)
```

Следующий пример демонстрирует фильтрацию исключений, соответствующих переполнению операции с плавающей запятой:

```
(GetExceptionCode() == STATUS_FLOAT_OVERFLOW ||
    GetExceptionCode() == STATUS_FLOAT_UNDERFLOW)
```

Еще один пример показывает использование тернарного оператора:

```
(GetExceptionCode() == STATUS_ACCESS_VIOLATION ? 1 : 0)
```

Во всех этих случаях возвращается значение 0 или 1, и эти фильтры используются для того, чтобы выполнить обработчик, например:

```
__try {
    char * address = 0;
    address[0] = 0;
} __except(GetExceptionCode() == STATUS_ACCESS_VIOLATION) {
    // обработка нарушения доступа к памяти
}
```

Здесь выполняется обработчик при ошибке доступа к памяти, в противном случае управление передается вышестоящему обработчику.

3.3.3. Использование фильтрующей функции

В этом случае вместо выражения подставляется функция, которая анализирует код ошибки исключения, а также дополнительную информацию об исключении, и возвращает одно из значений, 0 или 1.

Дополнительную информацию об исключении предоставляет еще одна функция — GetExceptionInformation. Она возвращает указатель на блок информации, описываемый структурой _EXCEPTION_POINTERS:

```
struct _EXCEPTION_POINTERS {
    EXCEPTION_RECORD * ExceptionRecord,
    CONTEXT * ContextRecord
};
```

Как видно, эта структура, в свою очередь, состоит из двух указателей на две другие структуры. Структура EXCEPTION_RECORD описывает сведения об исключении, а CONTEXT — о контексте исключения. Эта дополнительная информация используется для формирования сообщений.

Пример фильтрующей функции:

```
int filter(unsigned int code, _EXCEPTION_POINTERS * ep) {
    if (code == STATUS_INTEGER_DIVIDE_BY_ZERO) {
        // деление на ноль не обрабатываем
        return EXCEPTION_CONTINUE_SEARCH;
    } else {
        // другое исключение обрабатываем
        // анализируем информацию об исключении
        // например, флаги машинного слова
        DWORD flags = ep->ContextRecord->EFlags;
        return EXCEPTION_EXECUTE_HANDLER;
    }
}
```

Фильтрующая функция обрабатывает исключение, если оно не является целочисленным делением на ноль. Для этого она может использовать дополнительную информацию, предоставляемую структурами данных EXCEPTION_RECORD и CONTEXT. В этом случае фильтрующая функция возвращает EXCEPTION_EXECUTE_HANDLER, предписывая выполнить код обработчика. В случае возникновения ошибки деления на ноль функция возвращает EXCEPTION_CONTINUE_SEARCH, то есть отправляет исключение вышестоящему обработчику.

Фильтрующая функция используется следующим образом:

```
void main() {
    int x = 0, y = 0;
    __try {
        y = 5 / x;
    } __except( filter( GetExceptionCode(),
                GetExceptionInformation() ) ) {
        printf("Exception raised in main\n");
    }
}
```

Естественно, фильтрующая функция может принимать какие угодно дополнительные параметры, необходимые для анализа ситуации, и не принимать результат функции GetExceptionInformation, если это не требуется. Функция может быть сколь угодно сложной, но должна возвращать либо нулевое значение, либо единичное.

3.4. Раскрутка стека

Обработчики исключений могут располагаться в любых функциях программы. Поскольку одни функции могут вызывать другие функции, обработчики формируют иерархию, в которой исключение, возникшее в одной из вложенных функций, может вызвать срабатывание обработчика, расположенного выше в этой иерархии. Так как управление при этом переходит из одной функции в другую, нужен механизм, закрывающий кадры стека тех функций, которые неожиданно завершили свое выполнение. Этот механизм называется раскруткой стека (stack unwind).

В качестве примера рассмотрим следующую программу:

```
void f2(int * x) {
    __try {
        *x = 1 / *x;
    } __finally {
        *x = 1;
    }
}

void f1(int * x) {
    __try {
        f2(x);
    } __finally {
        *x = 2;
    }
}

void main() {
    int result = 0;
    __try {
        f1(&result);
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        printf("Exception raised\n");
    }
    printf("result = %d\n", result);
}
```

Не так просто предсказать результат ее работы.

Если выполнять пошаговую отладку этого кода, то после выполнения деления на ноль в функции `f2` управление будет передано обработчику функции `main`, который выведет сообщение, после чего будет выведен результат, равный двум. Выполнение блоков завершения, однако, не будет показано, хотя оба блока будут выполнены.

При вызове функций `f1` и `f2` в стеке были размещены их кадры, и они должны быть закрыты, чтобы не нарушить правильное функционирование стека. Однако, поскольку в этих функциях есть блоки завершения, они выполняются до закрытия кадров. В этом можно убедиться, если установить в них точки остановки.

3.5. Программные исключения

Функции могут выявлять исключительные ситуации, генерируемые не только аппаратурой. Если в результате анализа обрабатываемых данных функция обнаруживает какое-либо несоответствие, она может генерировать программное исключение. Это механизм является альтернативой механизму обработки ошибки на месте.

Программное исключение формирует функция RaiseException. Параметрами этой функции являются:

DWORD dwExceptionCode — код ошибки,

DWORD dwExceptionFlags — флаги,

DWORD nNumberOfArguments — число дополнительных параметров,

CONST ULONG_PTR* pArguments — дополнительные параметры.

Для генерирования исключения сначала следует сформировать код ошибки. При этом биты 31 и 30 должны указывать на серьезность ошибки, бит 29 должен быть равен 1, бит 28 равен 0, биты 27-16 должны указывать на одну из подсистем, определенных Microsoft, биты 15-0 содержат произвольный код, идентифицирующий ошибку. Например, если вы решили генерировать серьезную ошибку с номером 1, код ошибки будет иметь значение 0xE0000001, при этом подсистема не указана.

Параметр «флаги» может принимать либо нулевое значение, либо значение EXCEPTION_NONCONTIUABLE. Во втором случае фильтр обработчика не может возвращать значение EXCEPTION_CONTINUE_EXECUTION.

Параметры используются редко.

В качестве примера приведен код, генерирующий некоторое программное исключение при обнаружении ошибки алгоритмом:

```
int division(int x) {
    if (x == 0) {
        RaiseException(0xE0000001, 0, 0, 0);
    }
    return 1 / x;
}

void main() {
    int result = 0;
    __try {
        result = division(0);
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        printf("Exception raised 0x%X\n", GetExceptionCode());
    }
    printf("result = %d\n", result);
}
```

Выполнение этого кода генерирует исключение, код исключения выводится в консоль.

4. Реестр Windows

Реестр Windows — это системная база данных, содержащая различные параметры (настройки) операционной системы и прикладных программ. В первых версиях Windows параметры можно было сохранять в одном из следующих мест сохранения:

- системные файлы инициализации;
- файлы инициализации приложений;
- файл регистрационной базы данных reg.dat.

К системным файлам инициализации относятся файлы system.ini, win.ini, program.ini, control.ini и protocol.ini. Файл system.ini являлся основным хранилищем информации, относящейся к оборудованию. Файл win.ini предназначен для хранения информации о конфигурации операционной системы, а также некоторые настройки прикладных программ. Файл program.ini содержал начальные установки диспетчера программ (Windows Program Manager). Файл control.ini содержал настройки панели управления (Control Panel). Файл protocol.ini появился в первой сетевой системе Windows for Workgroups, и содержал сетевые настройки.

Файлы инициализации приложений, называемые также частными (private), предназначены для сохранения настроек приложений, таких, как расположение окна на экране и его размер, настройки панелей инструментов, список недавно использованных документов и т.п.

Файл регистрационной базы данных является прямым предшественником современного реестра. Первоначально он содержал единственный раздел HKEY_CLASSES_ROOT, используемый для поддержки OLE, технологии, которая только начинала развиваться, и для сопоставления расширений имен файлов документов с приложениями, которые могли эти документы обрабатывать.

В связи с развитием аппаратуры, системы Windows, и появлением множества приложений выявились недостатки файлов инициализации.

1. Часто требуется ручное редактирование файлов.
2. Нет четких правил хранения файлов инициализации.
3. Не поддерживается многопользовательская среда.
4. Не поддерживаются множественные аппаратные конфигурации.
5. Невозможно совместно использовать информацию.
6. Недостаточная структурированность информации.

Для устранения указанных недостатков в Windows NT 3.5 впервые появился реестр в его современном понимании. Он представлял собой централизованный источник конфигурационной информации, и обеспечивал возможность эффективного управления средой Windows NT. Заметим, что файлы win.ini и system.ini существуют до сих пор для обеспечения совместимости с 16-битными приложениями.

4.1. Файлы инициализации

Несмотря на недостатки файлов инициализации, в некоторых случаях они удобнее записей в реестре. Файл инициализации — это текстовый файл, параметр в котором записывается в виде «ключ = значение», а параметры разных приложений записываются в разных секциях, помечаемых названием приложения в квадратных скобках:

```
[AppName1]
param1=value1
param2=value2
[AppName2]
param1=value1
param2=value2
```

Различают системные и частные файлы инициализации. К системным файлам относится в первую очередь файл win.ini. Для записи и чтения информации из этого файла есть следующие функции.

Функция WriteProfileString записывает параметр в файл win.ini:

```
BOOL WriteProfileString(
    LPCTSTR lpAppName,     // название секции
    LPCTSTR lpKeyName,     // ключ параметра
    LPCTSTR lpString       // записываемая строка
);
```

Приложения также могут записывать свою информацию в этот файл, однако делать это в настоящее время не следует.

Для чтения информации из файла win.ini используются функции:

```
DWORD GetProfileString(
    LPCTSTR lpAppName,         // название секции
    LPCTSTR lpKeyName,         // ключ параметра
    LPCTSTR lpDefault,         // значение по умолчанию
    LPTSTR lpReturnedString,   // буфер для информации
    DWORD nSize                // размер буфера
);
UINT GetProfileInt(
    LPCTSTR lpAppName,  // название секции
    LPCTSTR lpKeyName,  // ключ параметра
    INT nDefault        // значение по умолчанию
);
```

Характерной особенностью этих функций является наличие значения по умолчанию, которое возвращается в случае, если запрашиваемый параметр отсутствует.

Старые приложения могут считывать из файла win.ini информацию о национальных особенностях, таких, как название страны, формат даты и времени, формат денежных сумм, название денежной единицы и т.п.

Например, информацию о десятичном разделителе при выводе вещественного числа можно получить следующим образом:

```
DWORD n = GetProfileString("intl", "sDecimal", ".", buf, MAXB);
```

Здесь intl — название секции (international), sDecimal — название параметра. По умолчанию функция возвращает точку, информация записывается в буфер buf, функция возвращает в n размер записанной информации, включая завершающий ноль.

В современной системе файл win.ini не содержит раздела intl, а вызов функции транслируется в чтение информации из раздела реестра HKEY_CURRENT_USER\Control Panel\International.

Здесь проявляется также недостаток файла инициализации хранить множественные конфигурации. Информация из раздела International относится к локализации операционной системы. Если нужно получить информацию о национальных особенностях для другой локализации, следует использовать функцию:

```
int GetLocaleInfo(
    LCID Locale,      // идентификатор локали
    LCTYPE LCType,    // константа запрашиваемой информации
    LPTSTR lpLCData,  // буфер для информации
    int cchData       // размер буфера
);
```

Следующий пример показывает, как получить десятичный разделитель при помощи этой функции:

```
DWORD n = GetLocaleInfo(locale, LOCALE_SDECIMAL, buf, MAXB);
```

Здесь locale — идентификатор локализации, LOCALE_SDECIMAL — константа, соответствующая названию параметра sDecimal.

Для чтения и записи информации в частный файл инициализации есть аналогичные функции. Следующая функция записывает строку:

```
BOOL WritePrivateProfileString(
    LPCTSTR lpAppName,  // название секции
    LPCTSTR lpKeyName,  // ключ параметра
    LPCTSTR lpString,   // строка для записи
    LPCTSTR lpFileName  // путь к файлу
);
```

Следующая функция считывает строку:

```
DWORD GetPrivateProfileString(
    LPCTSTR lpAppName,        // название секции
    LPCTSTR lpKeyName,        // ключ параметра
    LPCTSTR lpDefault,        // значение по умолчанию
    LPTSTR lpReturnedString,  // буфер для информации
    DWORD nSize,              // размер буфера
    LPCTSTR lpFileName        // путь к файлу
);
```

Следующая функцию считывает целое число:

```
UINT GetPrivateProfileInt(
    LPCTSTR lpAppName,  // название секции
    LPCTSTR lpKeyName,  // ключ параметра
    INT nDefault,       // значение по умолчанию
    LPCTSTR lpFileName  // путь к файлу
);
```

Существуют и другие функции для выполнения операций с файлом инициализации, здесь они не рассматриваются.

4.2. Структура реестра

Записи в реестре образуют иерархическую структуру, очень похожую на структуру файловой системы. В качестве устройств (дисков) в реестре выступают несколько узлов, называемых корневыми ключами (root keys). Существуют следующие корневые ключи (Windows XP):

HKEY_CLASSES_ROOT — записи OLE и COM, расширения файлов.

HKEY_CURRENT_USER — записи о профиле текущего пользователя.

HKEY_LOCAL_MACHINE — записи о профилях оборудования.

HKEY_USERS — записи о профилях пользователях.

HKEY_CURRENT_CONFIG — записи о текущем профиле оборудования.

Для этих ключей определены соответствующие константы, используемые при программировании. Часто используют короткие обозначения ключей HKCR, HKCU, HKLM, HKU, HKCC.

Корневые ключи являются точками отсчета путей в реестре, в конце которых находятся значения (параметры). Путь представляет собой подключи корневых ключей, точно так же, как в файловой системе одни каталоги являются подкаталогами других. Например, записи о национальных особенностях локализации текущего пользователя находятся в реестре на пути HKCU\Control Panel\International, где Control Panel является подключом (subkey) корневого ключа, а ключ International является подключом ключа Control Panel.

Реестр размещается на диске в виде нескольких файлов, называемых ульями (hives). Взаимосвязь между корневыми узлами и ульями не однозначна, хотя корневые узлы часто называют ульями, и в названия корневых ключей входит слово «улей» (HKEY — Hives Key). На самом деле ни один из корневых ключей не представляет физического улья.

Рассмотрим корневой ключ HKLM. В редакторе реестра он представляется состоящим из подключей HRDWARE, SAM, SECURITY, SOFTWARE, и SYSTEM, но на самом деле он является контейнером ульев SAM, SECURITY, SOFTWARE, SYSTEM, которые хранятся в файлах с такими же названиями каталоге %SystemRoot%\system32\config. Здесь же находятся файлы протоколов транзакций .log, копии файлов на момент завершения текстовой части установки .sav, а также некоторые другие файлы реестра.

Улей HARDWARE не является физическим, он формируется динамически при старте системы при помощи распознавателя оборудования, программы ntdetect.com. Такие динамические структуры реестра иногда называют временными (volatile) ульями.

Корневой ключ HKCU также не является ульем. Он является ссылкой на ключ HKU\userSID, где userSID — идентификатор безопасности (SID, security identifier) пользователя, осуществившего вход в систему. Типичный вид идентификатора безопасности в реестре — S-1-5-18.

Физически улья пользователей хранятся в виде файлов Ntuser.dat в каталогах %SystenDrive%\Documents and Settings\%Username%. Подключи ключей userSID вида UserSID_classes хранятся в виде файлов UsrClass.dat в каталоге %SystenDrive%\Documents and Settings\%Username%\Local Settings\Application Data\Microsoft\Windows.

Параметры ключа HKU\.Default используется до того, как в системе будет зарегистрирован пользователь. Физически параметры этого ключа записываются в файл Ntuser.dat в каталоге Default User каталога Document and Settings. Параметры подсистем операционной системы, выступающих в качестве пользователей, записываются в файлы Ntuser.dat каталогов Local Service (для SID S-1-5-19) и NetworkService (для SID S-1-5-20) каталога Document and Settings.

Параметры корневого ключа HKCC записываются в улье SYSTEM.

Корневой ключ HKCR является анахронизмом и оставлен в реестре для совместимости с 16-битными приложениями. Фактически это ссылка на подключ HKLM\Software\Classes, в который записываются общие данные, и на подключ HKCU\Software\Classes, куда записываются данные, специфичные для текущего пользователя, а ключ HKCU является динамическим объединением этих подключей.

В отличие от файлов инициализации, в реестр могут быть записаны разнообразные данные. Для каждого типа данных определена константа, которая используется при отображении данных в редакторе реестра и при программном создании записей.

Основные типы данных следующие.

REG_BINARY. Двоичные данные.

REG_DWORD. Целое число без знака (двойное слово).

REG_EXPAND_SZ. Строка символов, может содержать переменную, например, %SystemRoot%, которая заменяется значением при чтении.

REG_MULTI_SZ. Многострочное поле, разделитель строк нулевой знак, в конце всех строк два нулевых знака.

REG_SZ. Строка символов.

REG_NONE. Нет определенного типа.

REG_QWORD. 64-разрядное число без знака.

4.3. API реестра

Для программного выполнения операций с реестром используется API реестра, множество функций, название которых начинается с Reg. Следует использовать функции, в названии которых есть суффикс Ex. Аналогичные функции без суффикса предназначены для совместимости с 16-битными приложениями, они не используют, например, атрибуты безопасности. Рассмотрим основные функции реестра.

Ключ (подключ) создает функция RegCreateKeyEx:

```
LONG RegCreateKeyEx(
    HKEY hKey,                  // [in] дескриптор ключа отсчета
    LPCTSTR lpSubKey,           // [in] название создаваемого ключа
    DWORD Reserved,             // [in] зарезервировано, всегда 0
    LPTSTR lpClass,             // [in] класс ключа
    DWORD dwOptions,            // [in] специальные параметры
    REGSAM samDesired,          // [in] маска доступа к ключу
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // [in] наследование
    PHKEY phkResult,            // [out] возвращаемый дескриптор
    LPDWORD lpdwDisposition     // [out] буфер результата
);
```

В качестве дескриптора ключа отсчета может быть использована одна из констант корневых ключей (они описаны выше), или дескриптор ключа, открытого при помощи функции RegOpenKeyEx.

В качестве названия создаваемого ключа используется строка, описывающая либо одно название, либо путь к ключу. В качестве примера далее будем рассматривать путь Software\OTI\Regina\1.0, который можно создать во вспомогательном буфере следующим образом:

```
sprintf(buf, "Software\\%s\\%s\\%s", COMPANY, APPNAME, VERSION);
```

Здесь константа COMPANY задает название организации, APPNAME задает название приложения, VERSION — задает версию приложения. Такая последовательность ключей является рекомендуемой.

Класс ключа — это строка, описывающая тип создаваемого ключа. Этот параметр можно быть равен NULL, или строке, описывающей назначение ключа, например, «Application Global Data».

Специальные параметры определяют способ хранения создаваемого ключа. Используется одна из следующих констант.

REG_OPTION_NON_VOLATILE — ключ сохраняется в улье.

REG_OPTION_VOLATILE — временный ключ, хранится в памяти.

REG_OPTION_BACKUP_RESTORE — используется при архивировании и резервировании, здесь подробно не рассматривается.

Маска доступа определяет, какие операции разрешены для данного ключа. Маска может составляться как комбинация констант, приведенных в следующей таблице.

Таблица 4. Маски доступа

Константа	Разрешает
KEY_CREATE_LINK	создавать символическую ссылку
KEY_CREATE_SUB_KEY	создавать подключи
KEY_ENUMERATE_SUB_KEYS	перечислять подключи
KEY_EXECUTE	чтение
KEY_NOTIFY	получать уведомление об изменении
KEY_QUERY_VALUE	запрашивать данные
KEY_SET_VALUE	записывать данные
KEY_ALL_ACCESS комбинация ключей	KEY_QUERY_VALUE KEY_ENUMERATE_SUB_KEYS KEY_NOTIFY KEY_CREATE_SUB_KEY KEY_CREATE_LINK KEY_SET_VALUE
KEY_READ комбинация ключей	STANDARD_RIGHTS_READ KEY_QUERY_VALUE KEY_ENUMERATE_SUB_KEYS KEY_NOTIFY
KEY_WOW64_64KEY	открыть 64-битный ключ
KEY_WOW64_32KEY	открыть 32-битный ключ
KEY_WRITE комбинация ключей	STANDARD_RIGHTS_WRITE KEY_SET_VALUE KEY_CREATE_SUB_KEY

Атрибуты безопасности рассматриваются в другой главе пособия.

Возвращаемый дескриптор — переменная типа PHKEY, в которую функция записывает дескриптор созданного ключа. Буфер результата — это переменная типа DWORD, передаваемая по ссылке, в которую функция записывает значение одной из констант:

REG_CREATED_NEW_KEY — был создан новый ключ;

REG_OPENED_EXISTING_KEY — ключ существовал и был открыт.

Функция возвращает ноль в случае успеха, или значение, которое соответствует одному из кодов ошибки файла winerror.h. Ключ отсчета, если не является корневым ключом, должен быть открыт с маской доступа KEY_CREATE_SUB_KEY. Нельзя создавать подключи непосредственно в корневых ключах HKLM и HKU (структура реестра такова, что эти корневые ключи состоят из ульев, а улей создать нельзя).

Пример создания записи в реестре, путь задан в буфере buf:

```
PHKEY key = NULL;
DWORD dwDisposition = 0;
LONG lResult = RegCreateKeyEx(HKEY_LOCAL_MACHINE, buf, 0, NULL,
    REG_OPTION_NON_VOLATILE, KEY_READ, 0, key, &dwDisposition);
if (lResult) goto registry_error;
```

При этом создаются все ключи, указанные в пути, если их не существовало, этим ключам присваиваются одинаковые маски, функция возвращает дескриптор ключа, название которого является последним.

Ключ реестра открывает функция RegOpenKeyEx:

```
LONG RegOpenKeyEx(
    HKEY hKey,          // [in] дескриптор ключа отсчета
    LPCTSTR lpSubKey,   // [in] название открываемого ключа
    DWORD ulOptions,    // [in] зарезервировано, всегда 0
    REGSAM samDesired, // [in] маска доступа
    PHKEY phkResult     // [out] возвращаемый дескриптор ключа
);
```

Никаких особенностей функция не имеет. Пример использования:

```
PHKEY key = NULL;
LONG lResult = RegOpenKeyEx(HKEY_LOCAL_MACHINE, buf, 0,
    KEY_READ, key);
if (lResult) goto registry_error;
```

Данный пример открывает ключ с названием 1.0.

После того, как ключ создан или открыт, данные под ключом можно считывать или записывать.

Данные можно записать при помощи функции RegSetValueEx:

```
LONG RegSetValueEx(
    HKEY hKey,            // [in] дескриптор ключа
    LPCTSTR lpValueName, // [in] название параметра
    DWORD Reserved,       // [in] зарезервировано, всегда 0
    DWORD dwType,         // [in] константа типа данных
    CONST BYTE *lpData,   // [in] записываемые данные
    DWORD cbData          // [in] размер записываемых данных
);
```

Рассмотрим использование этой функции для записи разных типов данных. Предполагается, что key, — это ключ, открытый в предыдущем примере, путь к которому задан в буфере buf.

Следующий пример записывает число в параметр с именем Locale:

```
DWORD value = 1049;
LONG lResult = RegSetValueEx(key, "Locale", 0, REG_DWORD,
    (LPBYTE)&value, sizeof(value));
if (lResult) goto registry_error;
```

Следующий пример записывает строку в параметр с именем Name:

```
TCHAR value[] = "Студент ОТИ";
LONG lResult = RegSetValueEx(key, "Name", 0, REG_SZ,
    (LPBYTE)&value, strlen(value) + 1);
if (lResult) goto registry_error;
```

При записи многострочного значения нужно помнить, что в конце данных должно быть два нулевых байта.

Ключ может также иметь один параметр без имени типа REG_SZ. Такой параметр называется параметром по умолчанию. Для его записи вместо имени параметра указывается пустая строка или NULL.

Данные можно прочитать при помощи функции RegQueryValueEx:

```
LONG RegQueryValueEx(
    HKEY hKey,             // [in] дескриптор ключа
    LPCTSTR lpValueName,   // [in] название параметра
    LPDWORD lpReserved,    // [in] зарезервировано, всегда 0
    LPDWORD lpType,        // [in] буфер типа данных
    LPBYTE  lpData,        // [in, out] буфер значения
    LPDWORD lpcbData       // [in, out] размер прочитанных данных
);
```

Пример чтения числового параметра Locale:

```
DWORD dwRead = 0, dwType = 0, dwSize = sizeof(DWORD);
LONG lResult = RegQueryValueEx(key, "Locale", 0,
    &dwType, (LPBYTE)&dwRead, &dwSize);
if (lResult || dwType != REG_DWORD) goto registry_error;
```

Здесь в переменную dwType возвращается тип считанного значения, в переменную dwSize — размер, в переменную dwRead — результат.

Пример чтения строкового параметра Name сложнее. Сначала нужно узнать длину строки, а затем считать ее из ключа:

```
DWORD dwType = 0, dwSize = 0;
LONG lResult = RegQueryValueEx(key, "Name", 0,
    &dwType, NULL, &dwSize);
if (lResult || dwType != REG_SZ) goto registry_error;
LONG lResult = RegQueryValueEx(key, "Name", 0,
    &dwType, (LPBYTE)value_buffer, &dwSize);
if (lResult) goto registry_error;
```

Здесь value_buffer — предварительно созданный буфер достаточного размера для приема данных. Первый вызов функции записывает в переменную dwSize размер строки, который затем подставляется во второй вызов. В промежутке между вызовами переменную dwSize можно использовать для выделения буфера из кучи. Значение этой переменной учитывает завершающий строку нулевой знак (или два нулевых знака).

После завершения работы с открытым ключом его нужно закрыть при помощи функции RegCloseKey:

```
LONG RegCloseKey(HKEY [in] hKey);
```

Для удаления ключа используется функция RegDeleteKey:

```
LONG RegDeleteKey(
    HKEY hKey,          // [in] дескриптор открытого ключа
    LPCTSTR lpSubKey    // [in] название удаляемого подключа
);
```

4.4. Ассоциация файла документа с приложением

В корневой узел HKEY_CLASSES_ROOT записывается информация об OLE и COM, а также ассоциации типов файлов документов с приложениями, которые открывают, обрабатывают или печатают эти документы.

Пусть есть приложение regina.exe, расположенное в каталоге c:\, и файл документа этого приложения имеет расширение .regina. Для этого приложения следует создать в ключе HKCR следующие записи.

Первая запись связывает расширение с меткой приложения:

```
HKCR\.regina = "Regina"
```

Здесь Regina — это метка, записанная как значение по умолчанию в ключ .regina. Вторая запись соответствует метке приложения и содержит вложенные подключи:

```
HKCR\Regina = "OTI Regina"
HKCR\Regina\DefaultIcon = "c:\regina.exe,2"
HKCR\Regina\Shell\
HKCR\Regina\Shell\Open
HKCR\Regina\Shell\Open\Command = "c:\regina.exe %1"
```

Строка "OTI Regina" записана как значение по умолчанию ключа Regina. Она появится в проводнике как описание типа файла.

Строка "c:\regina.exe,2" записана как значение по умолчанию ключа DefaultIcon. Она описывает местонахождение значка, который будет сопоставляться с файлами приложения.

Строка "c:\regina.exe %1" записана как значение по умолчанию ключа Shell\Open\Command. Она описывает командную строку, которая открывает файл документа при помощи приложения. Приложение должно быть построено таким образом, чтобы открывать файл документа, если путь к файлу задан как первый параметр командной строки.

Если приложение имеет ключи командной строки для, например, печати документа, дополнительно может быть задана команда печати:

```
HKCR\Regina\Shell\Print
HKCR\Regina\Shell\Print\Command = "c:\regina.exe %1 /p"
```

Здесь предполагается, что /p — ключ, печатающий документ.

4.5. Редактор реестра

Для просмотра и редактирования записей в реестре в Windows есть приложение regedit.exe. В 64-битной Windows редакторов три. 64-битный редактор 64-битного реестра, regedit.exe, находится в каталоге %SystemRoot%\system32. 64-битный редактор 32-битного реестра regedit.exe находится в каталоге %SystemRoot%\system32\SYSWOW64. Здесь же находится 32-битный редактор 32-битного реестра regedt32.exe.

Редактор реестра отображает информацию в том виде, который более удобен для пользователя, но фактическая структура реестра другая, как было описано выше. Отображаемую структуру реестра создает подсистема ядра, реализующая реестр, — Configuration Manager (диспетчер конфигураций).

Пользоваться редактором реестра нужно очень осторожно. Случайно удаленные записи в большинстве случаев восстановить или очень сложно, или невозможно, хотя у системы предусмотрено немало средств восстановления. Для изменения каких-либо параметров в реестре нужно использовать консоли, которых в системе Windows немало. Они являются безопасным способом модификации реестра.

Параметры, записанные в ключах, можно редактировать, изменяя не только значения, но и тип, если вы понимаете, что делаете. Делать это можно для записей, сформированных своим приложением.

Информацию можно экспортировать из реестра в файл типа .reg. Для этого нужно выбрать ключ, выбрать в меню «Файл — Экспорт…», выбрать местоположение и название файла в стандартном диалоге сохранения, нажать кнопку «Сохранить».

Полученный файл является текстовым, его можно редактировать. В качестве примера приведено содержание файла, полученного из реестра после внесения в него записей из приведенных выше примеров:

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\OTI]
[HKEY_LOCAL_MACHINE\SOFTWARE\OTI\Regina]
[HKEY_LOCAL_MACHINE\SOFTWARE\OTI\Regina\1.0]
"Locale"=dword:00000419
"Name"="Студент ОТИ"
```

Полученный файл можно использовать для обратного внесения информации в реестр.

Выполнять операции с реестром из командной строки можно при помощи команды `reg`. Команда имеет много опций, получить их полный перечень можно, введя команду `reg /?`. Команда находится в том же каталоге, что и редактор реестра. В 64-битной Windows этих команд две, команда для 32-битного реестра находится там же, где и соответствующий редактор.

5. Атрибуты безопасности

Атрибуты безопасности были введены в системе Windows NT. Цель атрибутов безопасности — упорядочить доступ к важнейшим объектам ядра, таким, как файл или реестр, и к ресурсам приложений. Это дало возможность сертифицировать впоследствии эту систему как имеющую класс безопасности C2 (классификация министерства обороны США).

5.1. Модель управления доступом

Модель управления доступом содержит два базовых компонента:

- маркеры доступа, содержащие информацию о пользователе, осуществившем вход в систему;
- дескрипторы безопасности, описывающие доступ к защищаемому объекту.

Когда пользователь входит в систему, происходит его идентификация посредством имени и пароля. При успешном входе система формирует маркер доступа (access token). Каждый процесс, запущенный от имени этого пользователя, будет иметь копию маркера. Маркер доступа содержит идентификатор безопасности пользователя (SID, security identifier, называемый также системным идентификатором), который однозначно определяет пользователя и группу, к которой он принадлежит.

Маркер доступа содержит также список привилегий, которые даны пользователю или группе. Маркер используется для определения достаточности прав пользователя для получения доступа к защищаемому объекту, или для выполнения административной функции, такой, как запись в реестр.

Дескриптор безопасности (security descriptor) идентифицирует владельца объекта, и может также содержать списки доступа (ACL, access-control list) двух видов:

- дискреционный список доступа (DACL, discretionary access-control list), который определяет пользователей или группы пользователей, которым разрешен доступ к объекту;
- системный список доступа (SACL, system access-control list), который управляет тем, как система следит за попытками доступа к объекту.

Список доступа (ACL) содержит список записей управления доступом (ACE, access-control entry). Каждая запись определяет набор прав доступа и содержит идентификатор безопасности, идентифицирующий доверенное лицо (trustee), которому даны или запрещены определенные права. В качестве доверенного лица могут выступать учетная запись пользователя (user account), учетная запись группы (group account), и текущий сеанс (logon session). Заметим, что от лица пользователя в систему могут входить, например, сервисы.

5.2. Взаимодействие потока с защищаемым объектом

Взаимодействие потока с объектом отображает рисунок 3.

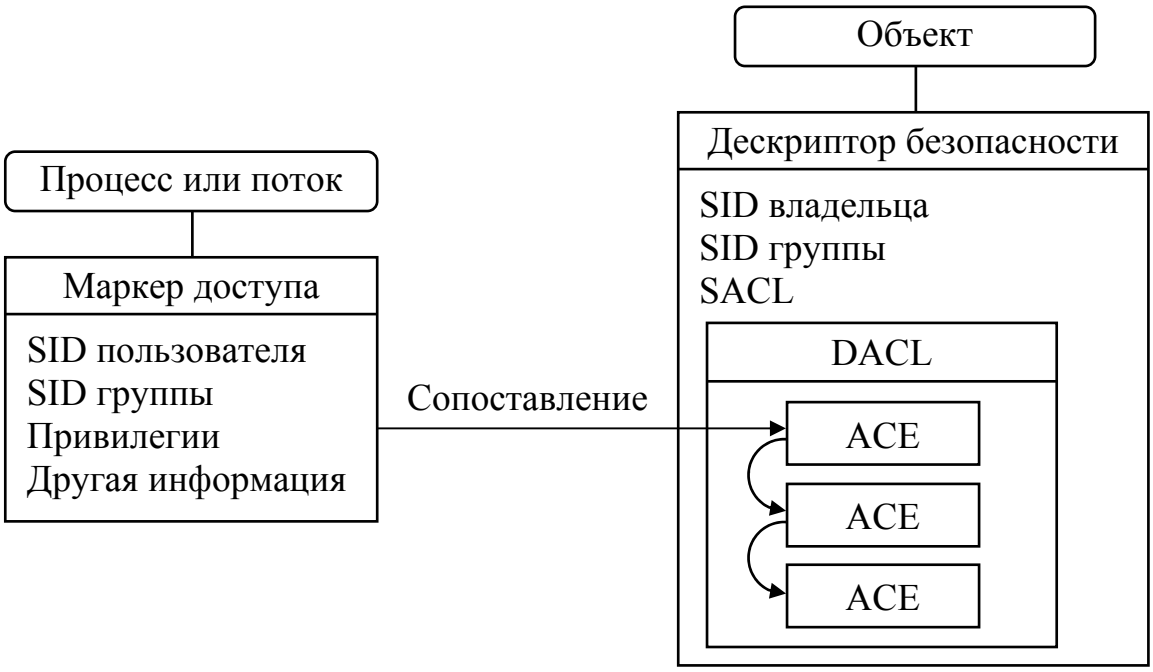


Рисунок 3 — Взаимодействие потока с защищаемым объектом

Когда поток пытается получить доступ к защищаемому объекту, система сопоставляет маркер доступа потока (или процесса, если у потока нет маркера) с дескриптором безопасности объекта. Маркер доступа содержит SID пользователя и группы, а дескриптор безопасности содержит дискреционный список доступа (DACL). Каждая запись управления доступом (ACE) сравнивается с идентификаторами безопасности маркера доступа. Если какой-либо идентификатор безопасности (SID) маркера является доверенным лицом, записанным в ACE, система дает потоку права этой ACE. Сопоставление продолжается до конца списка DACL. Если одна из ACE дает потоку право на чтение, а другая ACE дает право на запись, то поток получает право и на чтение, и на запись.

Одна ACE либо дает определенные права, либо запрещает определенные права, но не может одновременно что-то разрешать, а что-то запрещать. Первыми в списке DACL должны быть записи, запрещающие права доступа, а затем записи, разрешающие права доступа.

Если у объекта нет списка дискреционного доступа (DACL), система создает так называемый нулевой список доступа (null DACL), который разрешает полный доступ к объекту любому субъекту (пользователю, группе, программе). Нулевой список отличается от пустого списка, последний не дает никаких прав доступа к объекту.

5.3. Идентификатор безопасности

Идентификатор безопасности — это уникальная структура переменной длины, однозначно идентифицирующая доверенное лицо системы Windows NT. Идентификатор присваивается каждой учетной записи властью (authority), такой, как контроллер домена Windows, и сохраняется в специальной базе данных. Идентификатор безопасности состоит из следующих компонент:

- номер редакции структуры SID;
- 48-битный код власти, создавшей идентификатор;
- переменное число относительных идентификаторов (RID, relative identifier), которые однозначно определяют полномочия (subauthority) доверительного лица от имени власти, создавшей идентификатор.

Никакие два идентификатора безопасности не будут одинаковыми, даже если они имеют одинаковые полномочия, набор полномочий выдается властью единственный раз.

Для строкового представления идентификатор безопасности используется формат S-*R*-*I*-*S*-*S*…, где *R* обозначает номер редакции структуры SID, *I* идентифицирует власть, а *S* — полномочия.

Например, строка S-1-5-32-544 представляет SID локальной группы администраторов, где

5 (SECURITY_NT_AUTHORITY) — идентификатор власти,

32 (SECURITY_BUILTIN_DOMAIN_RID) — первое полномочие

544 (DOMAIN_ALIAS_RID_ADMINS) — второе полномочие.

В дополнение к уникальным идентификаторам, существуют хорошо известные идентификаторы (well-known SIDs), которые идентифицируют обобщенных пользователей или группы пользователей.

Следующие универсальные хорошо известные идентификаторы используют не только в операционной системе Windows NT.

Null SID	(S-1-0-0)	пустая группа
World	(S-1-1-0)	все пользователи
Local	(S-1-2-0)	пользователи локальной машины
Creator Owner ID	(S-1-3-0)	заменяется SID создателя
Creator Group ID	(S-1-4-0)	заменяется SID группы создателя

Следующие константы представляют власти (authority).

SECURITY_NULL_SID_AUTHORITY (0)	префикс S-1-0.
SECURITY_WORLD_SID_AUTHORITY (1)	префикс S-1-1.
SECURITY_LOCAL_SID_AUTHORITY (2)	префикс S-1-2.
SECURITY_CREATOR_SID_AUTHORITY (3)	префикс S-1-3.
SECURITY_NT_AUTHORITY (5)	префикс S-1-5.

Следующие константы определяют полномочия (RID) универсальных хорошо известных идентификаторов безопасности (властей).

SECURITY_NULL_RID (0)	для префикса S-1-0.
SECURITY_WORLD_RID (0)	для префикса S-1-1.
SECURITY_LOCAL_RID (0)	для префикса S-1-2.
SECURITY_CREATOR_OWNER_RID (0)	для префикса S-1-3.
SECURITY_CREATOR_GROUP_RID (1)	для префикса S-1-3.

Константа SECURITY_NT_AUTHORITY представляет власть, создающую идентификаторы безопасности, действительные только в операционной системе Windows NT. Следующие константы являются примерами полномочий, выдаваемых этой властью.

SECURITY_LOCAL_SYSTEM_RID (S-1–5–18) — учетная запись, используемая операционной системой.

SECURITY_BUILTIN_DOMAIN_RID (S-1–5–32) — встроенный домен.

Следующие константы являются примерами полномочий домена.

DOMAIN_ALIAS_RID_ADMINS	группа администраторов домена.
DOMAIN_ALIAS_RID_USERS	группа всех пользователей домена.

5.4. Права доступа

Права доступа к защищаемому объекту задаются маской, биты которой определяют конкретные привилегии.

Биты 0-15 определяют права доступа, специфичные для объекта.

Например, для доступа к реестру используются константы вида KEY_READ, код которых занимает младшие 16 бит маски прав доступа.

Биты 16-22 определяют стандартные права доступа, задаваемые константами DELETE (право удалять объект), READ_CONTROL (читать информацию дескриптора безопасности, исключая SACL), SYNCHRONIZE (использовать объект для синхронизации), WRITE_DAC (изменять DACL), WRITE_OWNER (изменять дескриптор безопасности).

Бит 23 определяет право доступа к SACL.

Биты 24-27 зарезервированы для системы.

Бит 28 определяет право GENERIC_ALL (сумма следующих прав).

Бит 29 определяет право GENERIC_READ.

Бит 30 определяет право GENERIC_WRITE.

Бит 31 определяет право GENERIC_EXECUTE.

5.5. Дескриптор безопасности

Для создания защищаемого объекта нужно описать атрибуты безопасности, структуру SECURITY_ATTRIBUTES, которая содержит ссылку на дескриптор безопасности, структуру SECURITY_DESCRIPTOR. Защищаемыми объектами являются маркеры доступа, файлы и каталоги NTFS, процессы и потоки, каналы (pipes), ключи реестра, сервисы, объекты синхронизации и некоторые другие.

Есть несколько путей создания дескриптора безопасности, использующие различные функции. Процесс в целом включает в себя:

- создание идентификаторов безопасности SID, описывающих пользователей или группы пользователей, которым будет разрешен или запрещен доступ к объекту;
- создание записей управления доступом ACE, определяющих конкретные права доступа к объекту для конкретных идентификаторов безопасности SID;
- создание дискреционного списка доступа DACL;
- прикрепление записей ACE к списку DACL;
- прикрепление списка DACL к дескриптору безопасности;
- прикрепление дескриптора безопасности к атрибутам безопасности. Атрибуты безопасности далее передаются функции, создающей защищаемый объект.

5.5.1. Создание идентификаторов безопасности

Идентификатор безопасности создает следующая функция:

```
BOOL AllocateAndInitializeSid(
    PSID_IDENTIFIER_AUTHORITY pIdentifierAuthority, // authority
    BYTE nSubAuthorityCount,  // число полномочий
    DWORD dwSubAuthority0,    // полномочие 0
    DWORD dwSubAuthority1,    // полномочие 1
    DWORD dwSubAuthority2,    // полномочие 2
    DWORD dwSubAuthority3,    // полномочие 3
    DWORD dwSubAuthority4,    // полномочие 4
    DWORD dwSubAuthority5,    // полномочие 5
    DWORD dwSubAuthority6,    // полномочие 6
    DWORD dwSubAuthority7,    // полномочие 7
    PSID *pSid                // возвращаемый SID
);
```

В качестве примера будем рассматривать предоставление полного доступа к ключу реестра группе администраторов, и доступа не чтение всем другим пользователям.

Первым параметром функции указывается структура, описывающая ведомство (власть), создающее идентификатор безопасности. Она задается следующим образом:

```
SID_IDENTIFIER_AUTHORITY AdminAuth = SECURITY_NT_AUTHORITY;
SID_IDENTIFIER_AUTHORITY WorldAuth = SECURITY_WORLD_SID_AUTHORITY;
```

Поскольку это структуры данных, нельзя непосредственно использовать константу типа SECURITY_NT_AUTHORITY как параметр функции. Эта константа определена следующим образом (файл winnt.h):

```
#define SECURITY_NT_AUTHORITY {0,0,0,0,0,5}
```

Полномочия задаются при помощи констант относительных идентификаторов RID. Создание SID для группы администраторов:

```
PSID psidAdmin = NULL;
if (!AllocateAndInitializeSid(&AdminAuth, 2,
     SECURITY_BUILTIN_DOMAIN_RID,
     DOMAIN_ALIAS_RID_ADMINS, 0, 0, 0, 0, 0, 0, &psidAdmin)) {
     goto init_sid_error;
}
```

Создание SID для всех других пользователей:

```
PSID psidWorld = NULL;
if (!AllocateAndInitializeSid(&WorldAuth, 1,
     SECURITY_WORLD_RID, 0, 0, 0, 0, 0, 0, 0, &psidWorld)) {
     goto init_sid_error;
}
```

5.5.2. Создание записей управления доступом

Первый способ использует массив структур EXPLICIT_ACCESS, каждая структура описывает одну запись ACE:

```
EXPLICIT_ACCESS ea[2];
ZeroMemory(&ea, 2 * sizeof(EXPLICIT_ACCESS));
```

Далее для каждого из SID описываем права доступа. Для группы администраторов предоставляем полный доступ, для группы всех других пользователей предоставляем доступ на чтение:

```
ea[0].grfAccessPermissions = KEY_ALL_ACCESS;
ea[0].grfAccessMode = GRANT_ACCESS;
ea[0].grfInheritance = NO_INHERITANCE;
ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[0].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
ea[0].Trustee.ptstrName = (LPTSTR)psidAdmin;
ea[1].grfAccessPermissions = KEY_READ;
ea[1].grfAccessMode = GRANT_ACCESS;
ea[1].grfInheritance= NO_INHERITANCE;
ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[1].Trustee.TrusteeType = TRUSTEE_IS_WELL_KNOWN_GROUP;
ea[1].Trustee.ptstrName = (LPTSTR)psidWorld;
```

5.5.3. Создание DACL

Следующая функция создаст новый список DACL:

```
DWORD SetEntriesInAcl(
     ULONG cCountOfExplicitEntries,           // число записей
     PEXPLICIT_ACCESS pListOfExplicitEntries, // буфер
     PACL OldAcl,                             // существующий ACL
     PACL *NewAcl                             // новый ACL
);
```

Новый список DACL создается, если третий параметр равен NULL:

```
if (!SetEntriesInAcl(2, ea, NULL, &pACL)) {
    goto set_ace_error;
}
```

Эта функция одновременно прикрепляет к списку DACL все записи ACE, определенные в структурах EXPLICIT_ACCESS.

Другой способ создать список DACL заключается в выделении памяти и инициализации структуры ACL. Расчет размера требуемой памяти достаточно сложный. Проще выделить заранее больший объем:

```
#define MAX_ACL_SIZE 1024
PACL pACL = NULL;
// выделяем память
if (!(pACL = (PACL)LocalAlloc(LPTR, ACL_BUFF_SIZE))) {
    goto alloc_acl_error;
}

// инициализируем ACL
if (!InitializeAcl(pACL, ACL_BUFF_SIZE, ACL_REVISION2)) {
    goto init_acl_error;
}
```

5.5.4. Прикрепление записей ACE к списку DACL

Первый способ использует функцию SetEntriesInAcl, он описан выше. Другой способ использует функцию AddAccessAllowedAce:

```
BOOL AddAccessAllowedAce(
    PACL pAcl,             // ACL
    DWORD dwAceRevision,   // номер редакции ACL
    DWORD AccessMask,      // маска доступа
    PSID pSid              // SID
);
```

Пример использования этой функции:

```
if (!AddAccessAllowedAce(pACL, ACL_REVISION2,
    KEY_ALL_ACCESS, psidAdmin)) {
    goto add_ace_failure;
}

if (!AddAccessAllowedAce(pACL, ACL_REVISION2,
    KEY_READ, psidWorld)) {
    goto add_ace_failure;
}
```

Для добавления записей, запрещающих доступ, в этом случае используется функция AddAccessDeniedAce. Если же используются структуры EXPLICIT_ACCESS, второй параметр задается константой DENY_ACCESS, что указывает на запрещение доступа. Напомним, что записи, запрещающие доступ, в DACL должны добавляться первыми.

5.5.5. Прикрепление DACL к дескриптору безопасности

Дескриптор безопасности является частью структуры атрибутов безопасности. Можно использовать элемент этой структуры, или отдельную переменную для дескриптора безопасности. Предположим, используется отдельная переменная sdPermissons, имеющая тип структуры SECURITY_DESCRIPTOR. Сначала нужно инициализировать ее при помощи функции InitializeSecurityDescriptor:

```
SECURITY_DESCRIPTOR sdPermissons;
if (!InitializeSecurityDescriptor(&sdPermissions,
    SECURITY_DESCRIPTOR_REVISION1)) {
    goto init_sd_error;
}
```

Функция SetSecurityDescriptorDacl прикрепляет DACL к дескриптору безопасности:

```
BOOL SetSecurityDescriptorDacl(
    PSECURITY_DESCRIPTOR pSecurityDescriptor, // дескриптор
    BOOL bDaclPresent,                        // DACL присутствует
    PACL pDacl,                               // список DACL
    BOOL bDaclDefaulted                       // DACL не явный
);
```

Второй параметр, равный TRUE, указывает, что список DACL прилагается. Четвертый параметр, равный FALSE, указывает, что список определен пользователем, а не получен стандартным механизмом:

```
if (!SetSecurityDescriptorDacl(&sdPermissions, TRUE, pACL, FALSE)) {
    goto set_sd_acl_error;
}
```

Дополнительно можно прикрепить к дескриптору безопасности SID владельца, например:

```
if (!SetSecurityDescriptorOwner(&sdPermissions, psidAdmin, 0)) {
    goto set_sd_owner_error;
}
```

Аналогично можно прикрепить SID группы при помощи функции SetSecurityDescriptorGroup.

Последнее, что нужно сделать — прикрепить дескриптор безопасности к атрибутам безопасности:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.bInheritHandle = FALSE;
sa.lpSecurityDescriptor = &sdPermissions;
```

Далее полученные атрибуты безопасности подставляются в функцию создания защищаемого объекта.

5.6. Маркер доступа

Маркер доступа получают для того, чтобы определить параметры безопасности пользователя, группы или объекта.

Маркер доступа потока возвращает функция OpenThreadToken:

```
BOOL OpenThreadToken(
    HANDLE ThreadHandle,  // дескриптор потока
    DWORD DesiredAccess,  // маска доступа
    BOOL OpenAsSelf,      // контекст безопасности процесса
    PHANDLE TokenHandle   // возвращаемый дескриптор маркера
);
```

Пример использования этой функции:

```
HANDLE hToken = NULL;
if (!OpenThreadToken(GetCurrentThread(), TOKEN_QUERY, 0, &hToken)) {
    // ошибка
}
```

Если в случае ошибки код ошибки равен ERROR_NO_TOKEN, поток не имеет маркера доступа. Тогда его можно получить у процесса при помощи функции OpenProcessToken:

```
BOOL OpenProcessToken(
    HANDLE ProcessHandle, // дескриптор процесса
    DWORD DesiredAccess,  // маска доступа
    PHANDLE TokenHandle   // возвращаемый дескриптор маркера
);
```

Пример использования этой функции:

```
if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) {
    // ошибка
}
```

Далее используется функция GetTokenInformation:

```
BOOL GetTokenInformation(
    HANDLE TokenHandle,            // дескриптор маркера
    TOKEN_INFORMATION_CLASS TokenInformationClass, // тип информации
    LPVOID TokenInformation,       // буфер для информации
    DWORD TokenInformationLength,  // размер буфера
    PDWORD ReturnLength            // требуемый размер буфера
);
```

Второй параметр указывает тип требуемой информации. Например, значение TokenGroups предполагает получение информации о группе, а значение TokenUser — о пользователе.

Чтобы узнать требуемый размер информации, эта функция вызывается с третьим параметром, равным NULL. Требуемый размер при этом возвращается последним параметром, а функция устанавливает код последней ошибки равным ERROR_INSUFFICIENT_BUFFER.

Пример использования этой функции сначала определяет требуемый размер буфера, затем выделяет память для информации о пользователе, затем получает собственно информацию:

```
DWORD dwSize = 0;
PTOKEN_USER ptu = NULL;
// запрашиваем размер иинформации
if (GetTokenInformation(hToken, TokenUser, 0, 0, &dwSize)) {
    goto get_size_error;
}
// проверяем правильность вызова
if (GetLastError() != ERROR_INSUFFICIENT_BUFFER) goto buffer_error;
// выделяем память для информации
if (!(ptu = LocalAlloc(LPTR, dwSize))) goto alloc_error;
// получаем иинформацию
if (!GetTokenInformation(hToken, TokenUser, ptu, dwSize, &dwSize)) {
    LocalFree(ptu);
    goto get_info_error;
}
```

Переменная ptu содержит SID пользователя в поле UserSid.

Кроме информации о пользователе и группе, при помощи этой функции можно получить информацию о DACL, о привилегиях и т.п.

6. Управление памятью

Каждому процессу Windows выделяется собственное виртуальное адресное пространство (ВАП). Для 32-разрядной системы размер ВАП составляет 4 Гб, для 64-разрядной системы размер ВАП представляет собой практически необозримый объем.

Адресное пространство процесса закрыто в том смысле, что только потоки процесса имеют к нему доступ. Несмотря на то, что ВАП представляет значительный объем, фактически это не физическая память, а выделенные диапазоны адресов. Виртуальное пространство процесса разбивается на разделы, как показано в следующей таблице.

Таблица 5. Разделы ВАП в 32-разрядной Windows XP

Раздел	Адрес	Размер
Для выявления нулевых указателей	0x00000000	64 Кб
Код и данные режима пользователя	0x00010000	2 Гб – 128 Кб
Закрытый	0x7FFF0000	64 Кб
Код и данные режима ядра	0x80000000	2 Гб

Первый раздел предназначен для выявления нулевых указателей. Когда программа обращается к памяти по такому указателю, происходит нарушение доступа.

Во втором разделе размещается код и данные режима пользователя. Эта часть адресного пространства является неразделяемой, то есть недоступной другим процессам. Заметим, что в предыдущих версиях системы Windows между этими разделами размещался раздел размером 4 Мб минус 4 Кб (отведенные под первый раздел), предназначенный для совместимости с программами MS-DOS и 16-разрядной Windows, а раздел процесса начинался с адреса 0x00400000. Примечательно, что параметр hInstance функции WinMain имеет именно это значение. В некоторых версиях Windows в раздел процесса имеет размер не 2, а 3 Гб. В основном это серверные версии системы. При компиляции программы в любом случае можно указать выделяемый размер 3 Гб. Для уменьшения размера раздела процесса в 64-разрядной системе до стандартных 2 Гб также используется ключ компиляции.

Закрытый третий раздел доступен только системе и предназначен для внутренних целей системы.

Наконец, оставшаяся память выделяется для кода и данных режима ядра и здесь располагаются код ядра, драйверы устройств, кэш-буферы ввода-вывода, области памяти, не сбрасываемые в файл подкачки, таблицы страниц и сегментов и т.п., так что едва хватает памяти для системных нужд.

6.1. Регионы в виртуальном адресном пространстве

В момент создания процесса его адресное пространство свободно. Чтобы использовать какую-нибудь его часть, нужно выделить в нем регионы при помощи функции VirtualAlloc. Операция выделения региона называется *резервированием*. При резервировании выделяемая память выравнивается по границе, равной параметру грануляция при выделения памяти (allocation granularity), который обычно составляет 64 Кб. Кроме того, размер региона принимается кратным размеру страницы. Размер страницы, как и грануляция, в принципе, зависит от типа процессора, но обычное значение равно 4 Кб.

Таким образом, если вы хотите зарезервировать регион размером 10 Кб, фактически будет выделено пространство размером 12 Кб.

Система может выделять регионы самостоятельно для обеспечения процесса. Например, в момент создания процесса выделяется регион для блока окружения процесса (process environment block, PEB). Для управления потоками создаются блоки окружения потоков (thread environment block, TEB), которые создаются и уничтожаются вместе с потоками.

Чтобы зарезервированный регион можно было использовать, нужно выделить физическую память и спроецировать ее на регион. Эта операция называется *передачей* физической памяти (committing physical storage), и ее выполняет также функция VirtualAlloc. Физическая память не обязательно выделяется всему региону, она может быть выделена только отдельным его частям (страницам). Например, при выделении памяти размером в 5 страниц, физическую память можно выделить только для первой и четвертой страницы региона.

В современных системах память имитируется за счет дискового пространства. При этом на диске создается так называемый страничный файл (paging file), который содержит виртуальную память. Без страничного файла система будет работать очень медленно. Кроме того, страничный файл может быть выделен для каждого диска в отдельности, что повышает производительность. По мере необходимости операционная система сбрасывает в страничный файл части оперативной памяти или подгружает их обратно. Физическая память при этом рассматривается как данные, хранимые в дисковом файле со страничной структурой.

Когда приложение передает физическую память региону адресного пространства, она выделяется из файла на диске. Заметим, что для уменьшения объема страничного файла система не загружает в него образы исполняемых файлов (EXE и DLL). Вместо этого файл проецируется в память и становится зарезервированным регионом адресного пространства, оставаясь на диске в месте своего расположения. Такой файл называется проецируемым в память (memory-mapped).

6.2. Атрибуты защиты региона

Следующие атрибуты могут быть присвоены регионам адресного пространства (таблица 6).

Таблица 6. Атрибуты защиты

Атрибут	Описание
PAGE_NOACCESS	Нет доступа
PAGE_READONLY	Только чтение
PAGE_READWRITE	Только чтение и запись
PAGE_EXECUTE	Только исполнение
PAGE_EXECUTE_READ	Запрещена запись
PAGE_EXECUTE_READWRITE	Любые операции
PAGE_WRITECOPY	Попытка записи создает копию страницы, исполнение запрещено
PAGE_EXECUTE_WRITECOPY	Попытка записи создает копию страницы, любая другая операция разрешена

Последние два атрибута защиты предназначены для экономии оперативной памяти и места в страничном файле.

Windows поддерживает механизм, при котором два и более процессов разделяют один и тот же блок памяти. Если, например, открыть несколько экземпляров блокнота, все они будут использовать один и тот же код и данные. Однако, если процесс начинает запись в общий блок памяти, система ищет свободную страницу в оперативной памяти, копирует страницу общего доступа в новую страницу, после чего отображает виртуальный адрес страницы общего доступа в процессе на новую страницу в оперативной памяти. Этой новой странице присваивается один из двух последних атрибутов защиты. Выделить регион памяти с этими атрибутами защиты при помощи VirtualAlloc нельзя.

Кроме перечисленных, есть еще три флага атрибутов защиты, которые комбинируются с перечисленными атрибутами защиты при помощи побитовой операции OR.

Флаг PAGE_NOCACHE отключает кэширование переданных страниц. Он предназначен в основном для разработчиков драйверов устройств, когда требуется манипулировать с буферами памяти.

Флаг PAGE_WRITECOMBINE также требуется разработчикам драйверов. Он позволяет объединять несколько операций записи на одно устройство в пакет, что увеличивает скорость передачи данных.

Флаг PAGE_GUARD позволяет приложению получать уведомление, когда в страницу производится запись (при этом используется механизм исключений). Флаг используется при создании стека потока.

6.3. Системная информация

Многие параметры операционной системы зависят от типа установленного в системе процессора. Если программа использует такие параметры, как грануляция при выделения памяти или размер страницы, эти параметры следует получать при помощи функции GetSystemInfo. Эта функция ничего не возвращает и имеет единственный параметр:

```
VOID GetSystemInfo(LPSYSTEM_INFO psinf);
```

Параметром является адрес структуры SYSTEM_INFO (winbase.h):

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId; // устаревший параметр, не используйте его
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

К памяти имеют отношение следующие поля.

dwPageSize — размер страницы в байтах.

lpMinimumApplicationAddress — минимальный адрес доступного каждому процессу адресного пространства.

lpMaximumApplicationAddress — минимальный адрес доступного каждому процессу адресного пространства.

dwAllocationGranularity — грануляция при выделении памяти.

Другие поля возвращают информацию о процессоре.

dwNumberOfProcessors — число процессоров.

dwActiveProcessorMask — битовая маска, показывающая, какие процессоры выполняют потоки (активны).

dwProcessorType — тип процессора (Windows 98).

wProcessorArchitecture — тип процессора (Windows 2000+).

wProcessorLevel — уточняет процессор, например, Pentium Pro или Pentium II (Windows 2000+).

wProcessorRevision — дополнительные подробности архитектуры процессора (Windows 2000+).

6.4. Статус виртуальной памяти

Текущее состояние памяти возвращает функция GlobalMemoryStatus:

```
VOID GlobalMemoryStatus(LPMEMORYSTATUS pmst);
```

Параметром является адрес структуры MEMORYSTATUS (winbase.h):

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORD dwTotalPhys;
    DWORD dwAvailPhys;
    DWORD dwTotalPageFile;
    DWORD dwAvailPageFile;
    DWORD dwTotalVirtual;
    DWORD dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Перед вызовом в поле dwLength нужно записать размер структуры.

В случае, если размер оперативной памяти или файла подкачки превышает 4 Гб, следует использовать функцию GlobalMemoryStatusEx:

```
VOID GlobalMemoryStatusEx(LPMEMORYSTATUSEX pmst);
```

Соответственно, используется структура данных MEMORYSTAUSEX. Элементы этой структуры имеют размер 64 бита.

Поля структур имеют следующие значения.

dwLength — показывает, насколько занята память.

dwTotalPhys — объем физической памяти в байтах.

dwAvailPhys — объем свободной физической памяти в байтах.

dwTotalPageFile — максимальный размер страничного файла.

dwAvailPageFile — свободный размер страничного файла.

dwTotalVirtual — объем виртуальной памяти процесса.

dwAvailVirtual — свободное адресное пространство процесса.

Для получения информации о состоянии участка памяти в данном процессе используется функция VirtualQuery:

```
VOID VirtualQuery(
    LPCVOID lpAddress,
    PMEMORY_BASIC_INFORMATION pbmi,
    DWORD dwLength
);
```

Аналогичная функция VirtualQueryEx возвращает информацию о состоянии участка памяти в другом процессе:

```
VOID VirtualQueryEx(
    HANDLE hProcess,
    LPCVOID lpAddress,
    PMEMORY_BASIC_INFORMATION pbmi,
    DWORD dwLength
);
```

Вторую функцию используют отладчики и системные утилиты. Обычное приложение использует первую функцию. Параметр lpAddress при вызове указывает на начало участка памяти. Информация о памяти возвращается в структуру MEMORY_BASIC_INFORMATION (winnt.h):

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Поля этой структуры имеют следующие значения.

BaseAddress — начальный адрес участка, округленный до ближайшего меньшего адреса, кратного размеру страницы.

AllocationBase — базовый адрес региона, включающий BaseAddress.

AllocationProtect — атрибут защиты региона при его резервировании.

RegionSize — размер региона, состоящего из группы страниц с одинаковыми атрибутами защиты, состоянием и типом (смежных страниц).

State — состояние смежных страниц. Имеет значение MEM_FREE, если память свободна, MEM_RESERVE, если память зарезервирована, или MEM_COMMIT, если память выдана. Если участок памяти свободен, поля AllocationBase, AllocationProtect, Protect и Type не определены, а если память только зарезервирована, то поле Protect не определено.

Protect — текущий атрибут защиты смежных страниц.

Type — тип физической памяти смежных страниц. Имеет значение MEM_IMAGE (образ памяти), MEM_MAPPED или MEM_PRIVATE. Последнее значение означает закрытую память процесса.

6.5. Использование виртуальной памяти

Виртуальная память является наиболее подходящим средством для работы с большим структурами данных. Функция VirtualAlloc резервирует регион адресного пространства и передает его физической памяти:

```
PVOID VirtualAlloc(
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwAllocationType,
    DWORD fdwProtect
);
```

Первый параметр — начальный адрес, но обычно указывают NULL, чтобы блок подходящего размера выбрала система. Возвращаемое значение адреса будет скорректировано с учетом грануляции.

Второй параметр задает требуемый размер в байтах.

Третий параметр задает действие — резервирование, или передачу физической памяти, или и то, и другое одновременно. Для резервирования передается MEM_RESERVE, для передачи MEM_COMMIT, при этом регион должен быть предварительно зарезервирован. Для резервирования и передачи используются обе константы, соединенные побитовой OR. Заметим, что регион, передаваемый физической памяти, не обязательно должен совпадать с зарезервированным регионом, но должен находиться внутри него (или совпадать с ним).

Последний параметр указывает требуемый атрибут защиты. Если память только зарезервирована, то допускаются только PAGE_NOACCESS, PAGE_GUARD, PAGE_NOCACHE, PAGE_WRITECOMBINE.

По окончании использования региона необходимо вернуть физическую память и освободить регион при помощи функции VirtualFree:

```
BOOL VirtualFree(
    LPVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwFreeType
);
```

Чтобы вернуть физическую память и освободить регион, требуется указать базовый адрес региона (полученный при вызове VirtualAlloc), размер, равный нулю, последний параметр MEM_RELEASE.

Чтобы вернуть физическую память, требуется указать адрес страницы, число возвращаемых байт, последний параметр MEM_DECOMMIT.

Функция VirtualProtect изменяет атрибуты защиты:

```
BOOL VirtualProtect(
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD flnewProtext,
    PDWORD pfloldProtect
);
```

Первые два параметра имеют тот же смысл, что и другие функции, с учетом грануляции и размера страницы. Новый атрибут защиты не может быть PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY. В последний параметр возвращается атрибут защиты, который был присвоен региону до вызова функции.

Иногда приложение занимает память на непродолжительное время. В этом случае можно сказать системе, чтобы она не записывала страницы в страничный файл. Такой отказ от страниц называется сбросом физической памяти (resetting of physical storage). Он выполняется функцией VirtualAlloc с передачей ей третьим параметром значения MEM_RESET.

Сброс страниц повышает быстродействие системы.

6.6. Проецируемые в память файлы

Проецирование файлов в память является удобным механизмом, позволяющим работать с физическим файлом на диске так, как будто он находится в оперативной памяти. Этот механизм не требует таких операций, как открытие, чтение или запись файла. Операционная система проецирует исполняемые файлы в память при их запуске, что позволяет экономить страничный файл, так как сам физический файл на диске становится частью виртуальной памяти.

Для проецирования файла в память нужно выполнить три шага:

- создать или открыть файл при помощи, например, CreateFile;
- создать проекцию файла при помощи CreateFileMapping;
- отобразить проекцию на память при помощи MapViewOfFile.

По завершении работы с файлом нужно выполнить также три шага:

- отменить отображение проекции на память (UnmapViewOfFile);
- освободить дескриптор объекта проекции;
- освободить дескриптор объекта файла.

Следующий пример кода показывает все эти действия.

```
// открываем файл
HANDLE hFile = CreateFile("C:\\test.txt",
    GENERIC_READ | GENERIC_WRITE, 0, 0,
    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
// получим размер файла
DWORD dwSize = GetFileSize(hFile, 0);
// создаем проекцию
HANDLE hFileMapping = CreateFileMapping(hFile,
    0, PAGE_READWRITE, 0, dwSize, 0);
// отоюражаем проекцию на память
TCHAR * pstrFile = (TCHAR*)MapViewOfFile(hFileMapping,
    FILE_MAP_WRITE, 0, 0, dwSize);
// выполняем операции с памятью, которые записываются в файл
pstrFile = _strrev((LPCVOID)pstrFile);
// отключаем проекцию от памяти
UnmapViewOfFile(pstrFile);
// закрываем объекты ядра
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Рассмотрим функцию CreateFileMapping:

```
HANDLE CreateFileMapping(
    HANDLE hFile,                       // описатель файла
    LPSECURITY_ATTRIBUTES lpAttributes, // атрибуты безопасности
    DWORD flProtect,                    // защита проекции
    DWORD dwMaximumSizeHigh,            // макс. размер проекции
    DWORD dwMaximumSizeLow,             // макс. размер проекции
    LPCTSTR lpName                      // имя объекта
);
```

Первый параметр — это дескриптор открытого файла.

Второй параметр задает атрибуты безопасности, как описано ранее.

Параметр flProtect задает атрибуты защиты. Он может быть одним из PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY. Допустимые флаги при этом SEC_NOCACHE и SEC_IMAGE. Первый флаг запрещает кэширование, второй указывает, что проецируется исполняемый файл PE (portable executable, EXE или DLL). Заметим, что если атрибут защиты не PAGE_READONLY, файл должен быть открыт с флагом GENERIC_WRITE.

Четвертый и пятый параметры указывают размер файла, который может составлять 2^{64} байт. Есть несколько вариантов задания размера. Если не предполагается изменять размер файла, то лучше всего задать нулевые значения. Если размер файла будет увеличен, задается предполагаемый его размер по окончании операций, при этом должен быть задан атрибут защиты PAGE_WRITE. Если файл фактически имеет нулевую длину, размер не может быть задан нулевыми значениями.

Последний параметр — это имя объекта проекции (если требуется).

После создания проекции нужно спроецировать ее на адресное пространство процесса при помощи функции MapViewOfFile:

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject,   // описатель проекции
    DWORD dwDesiredAccess,       // режим доступа
    DWORD dwFileOffsetHigh,      // смещение (старшая часть)
    DWORD dwFileOffsetLow,       // смещение (младшая часть)
    SIZE_T dwNumberOfBytesToMap  // число проецируемых байт
);
```

Первый параметр — это дескриптор созданной проекции.

Второй параметр задает тип доступа. Возможные варианты:

Тип доступа	Разрешено	Атрибут защиты
FILE_MAP_WRITE	Запись	PAGE_READWRITE
FILE_MAP_READ	Чтение	любой
FILE_MAP_ALL_ACCESS	Чтение и запись	PAGE_READWRITE
FILE_MAP_COPY	Чтение и запись	любой

В последнем варианте создается копия страницы.

Функция UnmapViewOfFile освобождает регион:

```
BOOL UnmapViewOfFile(PVOID pvBaseAddress);
```

Параметр этой функции должен совпадать со значением, полученным в вызове MapViewOfFile.

Механизм проецирования позволяет работать с одним и тем же файлом *одновременно нескольким приложениям*. Для этого приложения выполняют проецирование, используя одно и то же имя для проекции. В этом случае изменения, вносимые одним приложением, немедленно отображаются в проекции другого приложения.

6.7. Динамически распределяемая память

Динамически распределяемая память является удобным способом создания множества небольших блоков данных, например, при создании связных структур данных. Для этого приложению выделяется регион зарегистрированного адресного пространства, называемый кучей (heap). При использовании кучи нет необходимости учитывать грануляцию при выделения памяти и размер страниц. Однако выделение и освобождение памяти при этом происходит медленнее, куча подвержена фрагментации, а контроля над распределением памяти в куче у приложения нет.

При создании процесса создается стандартная куча размером 1 Мб. Используя ключ /heap компоновщика, размер может быть изменен. Кучу процесса используют в том числе функции API Windows, поэтому доступ к куче последовательный. У процесса может быть несколько дополнительных куч, которые создает и уничтожает программист. Стандартная куча создается и уничтожается автоматически.

Есть несколько причин создания дополнительных куч. Для примера пусть приложение использует два вида связных списков.

1. Защита компонентов

Элементы списков выделяются из одной и той же области памяти и перемешиваются в ней. Какие-либо нарушения в одном из списков могут привести к ошибкам в работе другого списка, и отладка становится затруднительной. Выделение для каждого из списка своей собственной кучи исключает нежелательное взаимовлияние элементов.

2. Более эффективное управление памятью

Списки могут иметь элементы разного размера, например, 32 и 24 байта. Если элементы обоих списков находятся в одной куче и перемешаны, при освобождении элементов память фрагментируется так, что некоторые ее блоки нельзя использовать повторно. Разные кучи для разных списков устраняет эту несовместимость.

3. Локализация доступа

Два разных списка, элементы которых перемешаны в одной куче, будут распределены по страницам таким образом, что элементы списков занимают несколько страниц каждый. Разные кучи способствуют тому, что элементы будут расположены более последовательно, и, вероятно, займут меньше страниц на каждый из списков. Операционной системе не придется непрерывно перезагружать страницы.

4. Быстрое освобождение памяти в куче

Быстро освободить память, занимаемую кучей, можно просто уничтожив ее, а затем, при необходимости, создав заново. Если все динамические структуры данных находятся в одной куче, высвобождение всех элементов какой-либо из структур займет немалое время.

Дополнительную кучу создает функция HeapCreate:

```
HANDLE HeapCreate(
    DWORD fdwOptions,
    SIZE_T dwInitialSize,
    SIZE_T dwMaximumSize
);
```

Первый параметр задает способ выполнения операций с кучей. Этот параметр может быть нулевым. Флаг HEAP_CREATE_ENABLE_EXECUTE указывает, что в куче может выполняться код. Флаг HEAP_NO_SERIALIZE предотвращает последовательное использование кучи, что иногда может привести к ее разрушению. Флаг HEAP_GENERATE_EXCEPTIONS указывает, что функции доступа к куче будут генерировать исключения в случае, например, нехватки памяти.

Второй параметр задает начальный размер кучи в байтах. Третий параметр задает максимальный размер. Если он не равен нулю, то будет создана куча именно такого размера. Если максимальный размер задать нулевым, то куча будет расти, пока это вообще возможно.

Функция возвращает дескриптор или NULL в случае неудачи.

Выделяет блок памяти из кучи функция HeapAlloc:

```
PVOID HeapAlloc(
    HANDLE hHeap,
    DWORD fdwFlags,
    SIZE_T dwBytes
);
```

Первый параметр — это дескриптор кучи, возвращаемый функцией HeapCreate, если используется дополнительная куча. Если используется стандартная куча, дескриптор возвращает функция GetProcessHeap.

Третий параметр задает требуемое количество байт.

Второй параметр задает способ использования блока, при этом возможно изменение способа, заданного функцией HeapCreate. Допустимые значения HEAP_GENERATE_EXCEPTIONS, HEAP_NO_SERIALIZE, они описаны выше, и HEAP_ZERO_MEMORY (заполнить блок нулями).

Если функция завершилась неудачно и разрешены исключения, то могут быть возбуждены исключения STATUS_NO_MEMORY (недостаточно памяти) и STATUS_ACCESS_VIOLATION (куча разрушена или параметры вызова неверны). В случае неудачи всегда возвращается NULL.

Изменить размер выделенного блока может функция HeapReAlloc:

```
PVOID HeapReAlloc(
    HANDLE hHeap,
    DWORD fdwFlags,
    PVOID pvMem,
    SIZE_T dwBytes
);
```

Параметр pvMem указывает на ранее выделенный блок.

Флаги включают в себя флаг HEAP_REALLOC_IN_PLACE_ONLY, запрещающий изменение положения выделенного ранее блока. Если блок нельзя расширить таким образом, функция завершается неудачно и возвращает NULL.

Истинный размер выделенного блока возвращает функция HeapSize:

```
SIZE_T HeapSize(
    HANDLE hHeap,
    DWORD fdwFlags,
    LPCVOID pvMem
);
```

Параметр fdwFlags может быть равен нулю или HEAP_NO_SERIALIZE.

Параметр pvMem указывает на блок.

Освобождает выделенный блок функция HeapFree:

```
BOOL HeapFree(
    HANDLE hHeap,
    DWORD fdwFlags,
    PVOID pvMem
);
```

Параметры и значения те же, что и у предыдущей функции. Блок должен быть выделен функцией HeapAlloc или HeapReAlloc.

Уничтожает дополнительную кучу функция HeapDestory:

```
BOOL HeapDestroy(HANDLE hHeap);
```

Уничтожить стандартную кучу нельзя.

Целостность кучи проверяет функция HeapValidate. Параметры этой функции те же, что и у предыдущих двух, поэтому здесь не приводятся. Если третий параметр равен нулю, проверяется целостность всех блоков кучи, иначе только конкретный блок.

Еще одна функция с такими же параметрами, функция HeapCompact, объединяет свободные блоки кучи в блоки большего размера.

Следующие две функции используются попарно:

```
BOOL HeapLock(HANDLE hHeap);
BOOL HeapUnlock(HANDLE hHeap);
```

Эти функции синхронизируют доступ. Если поток вызывает функцию HeapLock, он запрещает доступ к куче из других потоков до тех пор, пока не вызовет функцию HeapUnlock. Эти функции вызываются также другими функциями для работы с блоками и кучей в целом.

7. Библиотеки

Библиотеки — это наборы часто используемых функций, структур данных и ресурсов. Преимущества библиотек должны быть понятны, они позволяют многократно использовать один и от же код или же одни и те же ресурсы, такие, как диалоги, значки или строки. Различают статические и динамически подключаемые библиотеки.

Статические библиотеки (static libraries) имеют образ lib-файла, включаются в проект так же, как файлы кода, и компонуются в код приложения. Изменение статической библиотеки требует повторной компиляции приложения.

Динамические подключаемые библиотеки (dynamic-link libraries) имеют образ PE (portable executable), размещаются в dll-файлах, во время компоновки в код приложения записывается информация, необходимая для поиска DLL во время исполнения. Информацию о DLL содержит lib-файл, который создается одновременно с созданием DLL, этот файл требуется для построения приложения. Кроме того, требуется заголовочный файл, описывающий экспортируемые структуры данных. Во время выполнения операционная система ищет DLL и проецирует ее на адресное пространство процесса, после чего функции библиотеки или ресурсы становятся доступны приложению.

7.1. Статические библиотеки

Статические библиотеки просты. Создается проект статической библиотеки. Он отличается от проекта исполняемого файла отсутствием основной функции main. Далее обычным образом в модулях кода описываются функции. Для взаимодействия с кодом приложения функции и структуры данных описываются в заголовочном модуле. Статическая библиотека может содержать также переменные, константы, классы и объекты. При построении статической библиотеки создается lib-файл.

Недостаток статической библиотеки заключается в том, что использовать ее можно при программировании на том же языке, на котором написан код библиотеки, и, более того, желательно использовать один и тот же компилятор одной и той же версии.

7.2. Использование DLL

Динамически подключаемые библиотеки являются существенной составной частью операционной системы, в DLL содержатся функции API. В Windows kernel32.dll управляет памятью, процессами и потоками, user32.dll обеспечивает пользовательский интерфейс, gdi32.dll занимается графикой и выводом текста и т.д.. Есть несколько причин, по которым использование DLL имеет преимущество.

1. Несколько процессов одновременно используют одну и ту же DLL, проецируя ее на свое адресное пространство. При это экономится память и уменьшается свопинг.

2. Изменение функций в DLL не требует повторного построения приложения, как в случае со статической библиотекой (если при этом не изменяются название функции, количество, порядок и тип аргументов).

3. DLL обеспечиваются поддержкой после их распространения. Например, если DLL содержит функции управления мониторами, ее можно впоследствии дополнить функциями для мониторов, которые на момент создания DLL еще не существовали.

4. DLL могут быть использованы программами, написанными на разных языках программирования, при условии соблюдения правил вызова функций.

Недостаток использования DLL заключается в том, что файл приложения при этом не является само-достаточным, как в случае со статической библиотекой, а зависит от наличия и доступности DLL, при этом желательно еще иметь DLL определенной версии.

7.3. Основы DLL

Создание DLL также не является сложной задачей, хотя определенные сложности могут возникать. Порядок создания DLL следующий.

1. Создается проект динамически подключаемой библиотеки.

2. Создается заголовочный файл, содержащие описание типов, переменных и функций, при этом указывается, что экспортируется или импортируется, а также соглашение о типе вызова функций.

3. Описываются функции.

4. Строится проект.

В результате создаются lib-файл, файл зависимостей и сама DLL.

Для построения приложения необходим lib-файл, его нужно указать в свойствах проекта в разделе зависимостей компоновщика (линкера).

В отличие от статической библиотеки, проект DLL содержит файл dllmain.cpp, содержащий функцию DllMain, которая вызывается во время загрузки приложения и нужна для выполнения инициализации. В простейших случаях ее можно игнорировать или вообще удалить.

Самым важным при создании DLL является соглашение о типе вызова и учет декорация имен (mangling). Существует множество способов вызова функций (на уровне машинного кода). Самыми известными и используемыми при программировании на Си или C++ являются вызов в стиле Си, вызов в стиле Pascal и стандартный вызов.

При вызове в стиле Си параметры функции помещаются на стек задом наперед, а очистку стека (то есть удаление параметров) выполняет вызывающая функция (вызывающий код).

При вызове в стиле Pascal параметры помещаются на стек в прямом порядке, очистку стека выполняет вызываемая функция.

Стандартный вызов — это модификация вызова в стиле Pascal, в которой параметры помещаются на стек в обратном порядке.

Соглашения задают также, как возвращаются результаты функции, как используются регистры процессора и т.п.

При программировании на языке высокого уровня эти детали не учитываются в том смысле, что программист ничего сам на стек не помещает и стек не очищает, это делает за него компилятор, но в любом случае какое-то соглашение компилятором будет использовано, и при связывании приложения с DLL нужно учитывать это соглашение.

Второй момент касается декорации имен функций.

Компилятор Microsoft C, экспортируя C-функцию, искажает ее имя (в машинном коде). Он добавляет к имени знак подчеркивания, после имени знак @ и число, равное количеству передаваемых через стек байт. При этом подразумевается, что используется стандартное соглашение о вызове, помечаемое, при необходимости, модификатором __stdcall.

Компилятор Microsoft C++ декорирует имена всех функций и переменных гораздо сложнее (в чем легко убедиться, изучая листинг модуля на ассемблере). Кроме того, как написано в msdn, виртуальные функции не будут работать, если декорация отсутствует. Поэтому экспорт из DLL функций, и особенно классов, представляет собой непростую задачу, если предполагается использовать DLL в разных системах программирования, например, Microsoft Си и Borland Pascal. Всё будет проще, если приложение и DLL используют один и тот же компилятор.

Чтобы указать не декорированное экспортируемое имя, в проект DLL включают файл определения, def-файл:

```
LIBRARY "MYLIB"
EXPORTS
    MyFunc1 @1
    MyFunc2 @2
```

Здесь в первой строке задается имя библиотеки (это необязательная строчка файла определений). После строки EXPORTS указываются имена экспортируемых функций. Нумерация функций также не обязательна, если экспортируемые имена соответствуют именам функций.

Если код библиотеки пишется на C++, прототипы функций должны начинаться с extern "C", а если на Си, то делать этого не нужно. Это также необходимо для предотвращения декорации.

В msdn также сказано, что есть два способа описать экспортируемую функцию: либо при помощи def-файла, либо при помощи модификатора __declspec(dllexport). А импортируемые функции (в exe или dll модуле) должны описываться с модификатором __declspec(dllimport).

В файле DLL создается раздел экспорта, таблица, в которой перечислены имена экспортируемых функций. В файле EXE аналогичным образом создается раздел импорта, таблица, в которой указаны имена импортируемых функций и соответствующие им DLL. Посмотреть эти таблицы можно командой dumpbin с ключами -exports или -imports.

7.4. Связывание с DLL

7.4.1. Неявное связывание

При неявном (implicit) связывании EXE модуля с DLL загрузчик операционной системы сначала проецирует на адресное пространство процесса исполняемый модуль, затем анализирует его таблицу импорта. Далее загрузчик находит необходимые DLL и также проецирует их в память процесса. Поскольку в разделе импорта указано только название DLL, загрузчик ищет ее в следующих расположениях:

1) каталог EXE-файла.
2) каталог процесса.
3) системный каталог Windows.
4) основной каталог Windows.
5) каталоги переменной окружения PATH.

Если найти DLL не удается, приложение завершается.

После обнаружения и проецирования DLL на память процесса загрузчик корректирует код EXE, подставляя в вызовы экспортируемых функций фактические адреса. Заметим, что загружаемые DLL могут, в свою очередь, использовать функции других DLL, поэтому загрузчик исследует таблицу импорта каждой DLL и при необходимости загружает дополнительные DLL, при этом никакая DLL не загружается дважды.

7.4.2. Явное связывание

При явном (explicit) связывании EXE модуля с DLL программист явным образом загружает DLL, выполняет связывание, а по окончании использования DLL выгружает ее. Библиотеку загружает функция:

```
HMODULE LoadLibrary(LPCTSTR lpFileName);
```

Единственный параметр указывает путь к библиотеке. Возвращаемый дескриптор равен NULL, если модуль загрузить не удалось. Причина определяется функцией GetLastError.

Следующая функция также загружает библиотеку:

```
HMODULE LoadLibraryEx(
    LPCTSTR lpFileName,
    HANDLE hFile,
    DWORD dwFlags
);
```

Она отличается тем, что при помощи флагов можно задать поведение загрузчика при загрузке. Второй параметр здесь всегда NULL.

Следующая функция определяет, спроецирована DLL на адресное пространство процесса, или нет:

```
HMODULE GetModuleHandle(LPCTSTR lpModuleName);
```

Параметром является имя модуля (DLL или EXE). Эта функция также возвращает требуемый далее дескриптор.

Если загрузка библиотеки успешна, дескриптор используется для получения фактического адреса экспортируемой функции при помощи следующей функции:

```
FARPROC GetProcAddress(
    HMODULE hModule,
    LPCSTR lpProcName
);
```

Второй параметр указывает имя экспортируемой функции, которую необходимо вызвать. Написание функции должно с точностью до регистра соответствовать тому, что указано в def-файле DLL. Заметим, что тип этого параметра LPCSTR, то есть строка ANSI, потому что таблицы экспорта и импорта записываются в строках ANSI. Поэтому могут возникать непредвиденные исключительные ситуации, если, например, во время загрузки DLL функция DllMain использует строки Unicode.

В качестве второго параметра может быть также указан порядковый номер функции (в соответствии с таблицей экспорта и def-файлом). При этом используется макрос MAKEINTRESOURCE.

Функция GetProcAddress возвращает фактический адрес или NULL.

Последним действием при работе с явно связанной библиотекой является ее освобождение при помощи следующей функции:

```
BOOL FreeLibrary(HMODULE hModule);
```

Пусть, например, вызываемая функция экспортируется из sample.dll и имеет следующий прототип:

```
int MyFunc(int, int);
```

Последовательность действий при явном связывании следующая:

```
// описание типа вызываемой функции
typedef int (*EXPFUN)(int, int);
// загружаем библиотеку
HMODULE hModule = LoadLibrary("sample.dll");
// получаем адрес
EXPFUN addr = (EXPFUN)GetProcAddress(hModule, "MyFunc");
// выполняем вызов
int x = addr(1, 2);
// освобождаем библиотеку
FreeLibrary(hModule);
```

Здесь для простоты опущены проверки результатов вызовов функций, на самом деле делать этого нельзя, если не хотите, чтобы приложение привело к исключению, или того хуже, остановке компьютера.

7.5. Функция входа/выхода DLL

Функция DllMain (обратите внимание на регистр, Dll, а не DLL) предназначена для уведомления о событиях:

```
BOOL APIENTRY DllMain(HMODULE hModule, DWORD  ul_reason_for_call,
        LPVOID lpReserved) {
    switch (ul_reason_for_call) {
    case DLL_PROCESS_ATTACH: // загрузка DLL в память процесса
    case DLL_THREAD_ATTACH:  // запуск потока процесса
    case DLL_THREAD_DETACH:  // завершение потока процесса
    case DLL_PROCESS_DETACH: // выгрузка DLL из памяти процесса
        break;
    }
    return TRUE;
}
```

Функция при успешном развитии событий должна возвращать истину. Если она возвращает ложь, процесс завершается.

При возникновении одного из этих событий DLL может выполнять некоторую, достаточно простую инициализацию, например, открытие файла, создание кучи. При этом событие запуска первичного потока не поступает в DLL, вместо этого поступает событие загрузки DLL.

7.6. Переадресация вызовов

Некоторые библиотеки переадресуют вызовы своих экспортируемых функций другим DLL. Например, библиотека kernel32.dll переадресует вызов HeapAlloc в NTDLL.RtlAllocateHeap, то есть фактически функции HeapAlloc не существует и вызывается функция модуля NTDLL.dll. В дампе таблицы экспорта такие функции помечаются как «forwarded to».

Программист может задать переадресацию директивой pragma:

```
#pragma comment(linker, "/export:somefunc=somedll.someotherfunc")
```

7.7. Известные DLL и перенаправление

Некоторые DLL, поставляемые вместе с операционной системой, называются *известными*, потому что сведения о них записаны в реестр. Например, kernel32.dll является известной DLL. В реестре название файла DLL связывается с именем DLL. Например, для kernel32.dll именем является просто kernel32.Путь в реестре следующий:

HKLM\SYSTEM\CurrentControlSet\ControlSet\SessionManager|KnownDLLs

Функция LoadLibrary проверяет, указано ли расширение файла. Если нет, поиск DLL ведется обычным образом. Если да, расширение отбрасывается и ищется имя в разделе KnownDLLs.

С этим связано также перенаправление DLL. Чтобы загрузчик проверял сначала каталог приложения, а затем уже искал DLL в других местах, необходимо разместить в каталоге приложения файл с названием «имя-exe.exe.local», при этом содержимое файла не имеет значения. Например, если приложение называется myapp.exe, файл редиректа должен называться myapp.exe.local. При этом игнорируется путь, указанный в вызове LoadLibrary, и DLL ищется сначала в каталоге приложения. Известные DLL не могут быть перенаправлены таким образом. Все это нужно для того, чтобы связывать приложения с правильными DLL, так как часто DLL имеют несколько версий в разных расположениях.

Файл редиректа игнорируется, если есть файл манифеста.

7.8. Базовый адрес DLL

По умолчанию, базовый адрес EXE равен 0x400000, а базовый адрес DLL равен 0x10000000. Что происходит, если приложение использует не одну, а две DLL? Базовый адрес первой DLL будет 0x10000000, а для второй DLL система выберет другой адрес. Имеет ли это значение? Имеет, и оно заключается в том, что изменение базового адреса, который, кстати, указывается в заголовке PE и может быть задан программистом, влечет за собой переадресацию всех идентификаторов, используемых исполняемым файлом, во время загрузки по другому адресу, проще говоря, к замедлению выполнения. Поэтому выбор базового адреса имеет значение в некоторых случаях (когда приложение программируется с использованием нескольких собственных DLL).

Настройку DLL на новые адреса выполняет команда rebase. Здесь она не описывается, у нее есть справка. Команду следует использовать на таком этапе разработки, когда все DLL спроектированы и включены в приложение. Заметим, что системные DLL уже оптимизированы и всегда загружаются по оптимизированным адресам.

7.9. Модификаторы соглашения о вызове функции

Соглашение Си указывается модификатором __cdecl, стандартное соглашение указывается как __stdcall, это соглашение по умолчанию при программировании в Microsoft C++. Методы классов используют модификатор __thiscall, означающий, что указатель this передается как параметр метода в компиляторе GCC, и через регистр ECX в компиляторе Microsoft C++. Соглашение Pascal указывается модификатором __pascal, он не поддерживается в настоящее время (следует использовать WINAPI, результат этого макроса зависит от целевой платформы).

8. Взаимодействие между процессами

Адресные пространства процессов изолированы друг от друга, поэтому прямое общение процессов невозможно. Однако процессам часто требуется обмен информацией. Поэтому операционные системы могут реализовывать следующие механизмы межпроцессного взаимодействия.

1. Проецируемые в память файлы (file mapping).
2. Буфер обмена (clipboard).
3. Копирование данных (data copy).
4. Динамический обмен данными (dynamic data exchange, DDE).
5. Почтовые ящики (mailslots).
6. Каналы (pipes).
7. Очереди сообщений (message queue).
8. Вызов удаленной процедуры (remote procedure call, RPC).

Проецируемые в память файлы были описаны ранее. Здесь мы рассмотрим некоторые другие механизмы.

8.1. Буфер обмена

Буфер обмена — это разделяемая память, в которую одно приложение записывает информацию, а другое приложение считывает её. Обычный пользователь использует буфер обмена сочетаниями клавиш Ctrl+C, Ctrl+X, Ctrl+V. Информация в буфере имеет *формат буфера обмена* (*clipboard format*). Есть предопределенные форматы, описанные в файле winuser.h (приведены форматы, используемые DDE):

```
/* Predefined Clipboard Formats */
#define CF_TEXT             1
#define CF_BITMAP           2
#define CF_METAFILEPICT     3
#define CF_SYLK             4
#define CF_DIF              5
#define CF_TIFF             6
#define CF_OEMTEXT          7
#define CF_DIB              8
#define CF_PALETTE          9
#define CF_PENDATA          10
#define CF_RIFF             11
#define CF_WAVE             12
#define CF_UNICODETEXT      13
#define CF_ENHMETAFILE      14
```

Приложения могут регистрировать свои собственные форматы данных при помощи функции:

```
UINT RegisterClipboardFormat(LPCTSTR lpszFormatformat);
```

Любое приложение Windows может использовать буфер обмена. Например, таблицу Microsoft Excel можно вставить в документ Microsoft

Word или наоборот, таблицу Microsoft Word в Microsoft Excel. Нас интересует, в первую очередь, как использовать буфер обмена программно.

Следующие две парные функции открывают и закрывают буфер обмена, препятствуя его использованию другими приложениями:

```
BOOL OpenClipboard(HWND hWndNewOwner);
BOOL CloseClipboard(VOID);
```

Параметр hWndNewOwner — это дескриптор окна. При этом буфер связывается с потоком окна, другие приложения ждут, пока буфер будет закрыт. Этот параметр может быть равен NULL.

Следующие функции предназначены для очистки буфера, записи информации в буфер и извлечения информации из буфера:

```
BOOL EmptyClipboard(VOID);
HANDLE SetClipboardData(
    UINT uFormat, // формат данных
    HANDLE hMem   // дескриптор данных
);
HANDLE GetClipboardData(UINT uFormat);
```

Перед тем, как извлечь из буфера информацию, нужно убедиться в наличии в буфере требуемого формата при помощи функции:

```
BOOL IsClipboardFormatAvailable(UINT format);
```

Последовательность записи информации в буфер показывает следующий пример кода:

```
int main() {
    if (!OpenClipboard(NULL)) return 0;
    if (!EmptyClipboard()) goto closec;
    HGLOBAL hGlobal = GlobalAlloc(GPTR, 13);
    if (!hGlobal) goto closec;
    strcpy((char*)hGlobal, "1234567890\r\n");
    SetClipboardData(CF_TEXT, hGlobal);
    GlobalFree(hMem);
closec:
    CloseClipboard();
}
```

Здесь в буфер записывается 10 цифр, затем символы CR и LF (этого требует формат CF_TEXT) и нулевой байт в конце (он тоже часть формата CF_TEXT). Заметим, что память для информации должна выделяться только при помощи GlobalAlloc.

Заметим, что пошаговое выполнение кода консольного приложения возможно не всегда. Нужно запустить программу на выполнение, после чего в каком-нибудь приложении ввести Ctrl+V.

В буфер обмена записывают информацию во всех возможных ее форматах, начиная со сложных форматов, и заканчивая простыми. Приложение, считывающее информацию, выбирает наиболее подходящий

ему формат. Таким образом, буфер может хранить несколько форматов одновременно. Буфер обмена может также выполнять преобразования одних форматов в другие.

Следующий пример кода показывает, как получить информацию из буфера обмена (консольное приложение):

```
int main(void) {
    if (!IsClipboardFormatAvailable(CF_TEXT)) return 0;
    if (!OpenClipboard(NULL)) return 0;
    HGLOBAL hMem = GetClipboardData(CF_TEXT);
    if (!hMem) goto closec;
    printf("%s\n", (char*)hMem);
closec:
    CloseClipboard();
}
```

Используется этот тест следующим образом. Копируем текст программы при помощи Cttl+A, Ctrl+C, запускаем программу Ctrl+F5. Весь текст будет выведен в консоль.

В приведенных примерах для простоты область памяти hGlobal перед записью в нее данных не блокируется. Для тестового приложения этот код сработает, в реальных приложениях данные нужно блокировать при помощи функции GlobalLock перед любой операцией с данными в разделяемой памяти, а после операций данные необходимо разблокировать при помощи функции GlobalUnlock.

Пусть есть некая структура данных DATA, которую необходимо передать через буфер обмена или DDE. Последовательность действий при записи в нее информации поясняет следующий фрагмент кода:

```
struct DATA {
    DWORD dwSize;
    char Value[64];
};
DATA FAR* lpData;
HGLOBAL hGlobal;
void some_function() {
    hGlobal = GlobalAlloc(GMEM_MOVEABLE, sizeof(DATA));
    if (!hGlobal) . . .
    // блокируем данные
    lpData = (DATA FAR*)GlobalLock(hGlobal);
    if (!lpData) . . .
    // записываем данные
    lpData->dwSize = data_size;
    lstrcpy((LPSTR)lpData->Value, data_buffer);
    lstrcat((LPSTR)lpData->Value, (LPSTR)"\r\n");
    // разблокируем данные
    GlobalUnlock(hGlobal);
    // передаем hGlobal механизму обмена данными
    // кто-то освобождает hGlobal
}
```

Запись в буфер обмена информации, не подходящей ни под один из предопределенных форматов, отличается только тем, что нужно зарегистрировать формат. Функции RegisterClipboardFormat передается произвольная строка, описывающая формат, в ответ возвращается число, которое используется наравне с предопределенными идентификаторами. Оба приложения используют одну и ту же строку.

8.2. Копирование данных

Это совсем простой механизм, позволяющий передавать произвольные структуры данных из одного оконного приложения в другое.

Передающее приложение выполняет следующие действия.

1. Создает необходимую структуру данных, заполняет ее данными.
2. Заполняет специальную структуру данных COPYDATASTRUCT.
3. Посылает сообщение WM_COPYDATA окну приемника, прикрепляя специальную структуру как параметр lParam.

Следующий пример кода показывает эту последовательность:

```
struct mydata { char buffer[MAX_PATH] } data;
strcpy(data.buffer, "0123456789");
COPYDATASTRUCT cd;
cd.dwData = 1;
cd.cbData = strlen(data.buffer);
cd.lpData = (PVOID)&data;
SendMessage(hWnd2, WM_COPYDATA, (WPARAM)hWnd1, (LPARAM)(LPVOID)&cd);
```

В структуре COPYDATASTRUCT три элемента. Поле lpData содержит указатель на передаваемую структуру данных. Поле cbData содержит размер данных. Поле dwData содержит дополнительную информацию типа указателя на LONG. Оно используется в примере как команда, обозначающая действие, которое предполагается выполнить с данными.

Приемное окно, получив сообщение WM_COPYDATA, извлекает указатель на структуру данных, и далее извлекает информацию:

```
PCOPYDATASTRUCT pcd = (PCOPYDATASTRUCT)lParam;
strcpy(buffer, (char*)pcd->lpData);
```

8.3. Динамический обмен данными

Этот механизм устанавливает связи между оконными приложениями, позволяя пересылать и синхронизировать данные при помощи стандартной передачи сообщений Windows. Для упрощения этого процесса в Windows есть библиотека DDEML (DDE Management Library). Это DLL, используемая приложениями для разделяемых данных. Механизм DDE основан на сообщениях, атомах и указателях. DDEML использует функции и строковые идентификаторы. Эти два разных механизма являются совместимыми друг с другом.

Говорят, что два приложения, осуществляющие динамический обмен данными, ведут DDE разговор (conversation). Приложение, инициирующее разговор, называют DDE клиентом, приложение, отвечающее клиенту, называют DDE сервером. Приложение может вести несколько разговоров, и быть и клиентом и сервером одновременно.

Разговор происходит между двумя окнами. Каждое окно может участвовать ровно в одном разговоре, и это может быть главное окно, окно, связанное с документом, или специальное невидимое окно.

Протокол DDE рассматривает динамические данные как иерархию из трех уровней: приложение, тема (topic) и имя данных. Разговор уникально определяется названием приложения и темой. Данные, пересылаемые во время разговора, относятся к теме, и имеют стандартный или пользовательский *формат буфера обмена*.

Приложения обязаны вести системную тему, обеспечивающую участников разговора контекстной информацией. При этом данные должны иметь формат CF_TEXT, а элементы данных разделяются табулятором. Элементами системной темы могут являться форматы данных (Formats), справочные данные (Help), форматы подтверждений (ReturnMessages), состояния приложения типа готов, занят и тому подобное (Status), список поддерживаемых системных элементов (SysItems), список элементов тем (TopicItemList), список тем (Topics).

После установления разговора клиент может установить одну или несколько постоянных связей с сервером (permanent data link). Эта связь является механизмом уведомления клиента об изменении данных. Есть два варианта постоянной связи: теплая (warm) и горячая (hot).

При теплой связи клиенту посылается уведомление об изменении данных, после чего клиент запрашивает их. При горячей связи сервер посылает не уведомление, а изменившиеся данные.

Есть всего 9 сообщений DDE, приведенных в таблице 7.

Таблица 7. Сообщения DDE

Сообщение	Описание
WM_DDE_ACK	Уведомление о получении сообщения
WM_DDE_ADVISE	Установление постоянной связи
WM_DDE_DATA	Посылка данных клиенту
WM_DDE_EXECUTE	Посылка серверу списка команд
WM_DDE_INITIATE	Установление разговора
WM_DDE_POKE	Посылка данных серверу
WM_DDE_REQUEST	Запрос серверу послать данные
WM_DDE_TERMINATE	Завершение разговора
WM_DDE_UNADVISE	Завершение постоянной связи

Сообщения WM_DDE_INITIATE и WM_DDE_ACK посылают при помощи SendMessage, другие — при помощи PostMessage. Параметры: HWND окна приемника, сообщение, HWND окна получателя, аргументы.

Типичный разговор состоит из следующих событий.

1. Клиент инициирует разговор, сервер подтверждает.
2. Приложения обмениваются данными одним из способов:
- сервер посылает данные в ответ на запрос клиента;
- клиент посылает серверу запрашиваемые данные;
- клиент просит сервер уведомлять об изменении данных (warm);
- клиент просит сервер посылать данные при их изменении (hot);
- сервер выполняет команды по запросу клиента.
3. Клиент или сервер завершает разговор.

Если приложение ведет несколько разговоров одновременно, оно обязано обрабатывать сообщения каждого разговора строго в порядке их поступления, но может чередовать разговоры произвольно.

Механизм DDE похож на механизм Data Copy. На самом деле DDE — это протокол, который нужно выдерживать для того, чтобы общаться с приложениями, такими, как Microsoft Excel, которые этот протокол реализуют. Он включает в себя описанные выше сообщения, действия при получении сообщения определенного типа, специальную структуру данных DDEDATA, и порядок упаковки параметров. Поэтому при разработке приложения, которое не предполагает общения с приложениями, реализующими протокол, вести обмен можно так, как удобно.

8.3.1. Атомы

Элементами протокола являются приложение, тема и данные. Отдельный элемент идентифицируется строкой. В протоколе вместо строк используются атомы (тип ATOM), которые являются целыми 16-битными числами. В системе есть глобальные хэш-таблицы атомов, приложение может создавать локальные таблицы. Например, для DDE существует своя глобальная таблица атомов, для классов окон — своя.

Цель таблиц атомов — упростить работу со строками, заменив их числами, при этом никакие строки не дублируются, и одинаковым строкам соответствуют одинаковые атомы.

Различают числовые и строковые атомы. Числовые атомы имеют значение от 1 до 0xBFFF, строковые — от 0xC000 до 0xFFFF.

Добавляет атом в глобальную таблицу функция GlobalAddAtom:

```
ATOM GlobalAddAtom(LPCTSTR lpString);
```

Размер строки не должен превышать 255 знаков (плюс нулевой).

Удаляет атом из глобальной таблицы функция GlobalDeleteAtom:

```
ATOM GlobalDeleteAtom(ATOM nAtom);
```

Строку, соответствующую атому глобальной таблицы, возвращает функция GlobalGetAtomName (для числового атома возвращается #число):

```
UINT GlobalGetAtomName(
    ATOM nAtom,       // атом
    LPTSTR lpBuffer,  // буфер для строки
    int nSize         // размер буфера
);
```

Находит строку в глобальной таблице функция GlobalFindAtom:

```
ATOM GlobalFindAtom(LPCTSTR lpString);
```

Строковые атомы отличаются от числовых атомов тем, что для них в таблице атомов ведется учет ссылок, поэтому эти атомы можно уничтожать, не рискуя уничтожить сами строки.

8.3.2. Упаковка параметров

К сообщениям DDE прикрепляются два параметра, разные для разных сообщений, которые упаковываются функцией PackDDElParam, и передаются как параметр lParam функций SendMessage и PostMessage:

```
LPARAM PackDDElParam(
    UINT msg,      // сообщение DDE
    UINT_PTR uiLo, // младшее слово lParam
    UINT_PTR uiHi  // старшее слово lParam
);
```

Когда сообщение прибывает к получателю, параметры извлекает функция UnpackDDElParam:

```
BOOL UnpackDDElParam(
    UINT msg,          // сообщение DDE
    LPARAM lParam,     // lParam сообщения
    PUINT_PTR puiLo,   // младшее слово lParam
    PUINT_PTR puiHi    // старшее слово lParam
);
```

Как показывает многолетний опыт, переменные, в которые извлекаются параметры, не должны находиться в области видимости обратно вызываемой функции (функции обработки оконных сообщений).

Параметр удаляет функция FreeDDElParam:

```
BOOL FreeDDElParam(
    UINT msg,      // сообщение DDE
    LPARAM lParam  // lParam сообщения
);
```

Если сообщение не удалось отправить, функцию вызывает отправитель. Если сообщение поступило, функцию вызывает получатель.

Для повторного использования параметра нужно вызвать функцию ReuseDDElParam, которая здесь не описывается.

8.3.3. Структуры данных

В DDE используются несколько структур данных, разных для разных случаев. Для пересылки данных используется структура DDEDATA:

```
typedef struct {
  unsigned short unused:12,
  fResponse:1,
  fRelease:1,
  reserved:1,
  fAckReq:1;
  short cfFormat;
  BYTE Value[1];
} DDEDATA;
```

Структура содержит флаги, формат и данные. Данные записываются в поле Value. Флаг fResponse, равный 1, указывает, что данные посылаются в ответ на сообщение WM_DDE_DATA, иначе в ответ на сообщение WM_DDE_ADVISE. Флаг fRelease, равный 1, указывает, что освобождение памяти выполняет получатель сообщения. Флаг fAckReq, равный 1, указывает, что требуется подтверждение (то есть сообщение WM_DDE_ACK). Поле cfFormat — это формат буфера обмена, в котором пришли данные.

Кроме этой структуры, для разных видов сообщений используются структуры DDEACK, DDEADVISE, DDEPOKE, HSZPAIR.

8.3.4. Протокол

Все остальное, что есть в DDE — это порядок действий.

Рассмотрим пример, в котором инициируется разговор.

Клиент посылает сообщение WM_DDE_INITIATE, прикрепляя к нему атом названия приложения и атом названия темы:

```
atomApplication = GlobalAddAtom(szApplication);
atomTopic = GlobalAddAtom(szTopic);
SendMessage(HWND_BROADCAST, WM_DDE_INITIATE,
     (WPARAM)hWnd, MAKELONG(atomApplication, atomTopic));
GlobalDeleteAtom(atomApplication);
GlobalDeleteAtom(atomTopic);
```

Сообщение посылается всем, на что указывает значение первого параметра функции SendMessage (HWND_BROADCAST), при этом в ответ может быть получено несколько подтверждений от потенциальных серверов. Параметр hWnd — это дескриптор окна клиента.

Ответные действия сервера такие же, только посылается сообщение WM_DDE_ACK. Сервер может послать несколько подтверждений с разными темами. Дескриптор окна клиента сервер извлекает из сообщения WM_DDE_INITIATE. Клиент, в свою очередь, узнаёт дескриптор окна сервера, получив подтверждение в сообщении WM_DDE_ACK.

Если разговор установлен, клиент может выполнять запросы.

Для получения одного элемента данных клиент посылает серверу сообщение WM_DDE_REQUSET, указывая формат и атом имени данных:

```
LPARAM lParam = PackDDElParam(WM_DDE_REQUEST, CF_TEXT, atom);
PostMessage(W2, WM_DDE_REQUEST, (WPARAM)W1, lParam);
```

Здесь W1 — дескриптор окна клиента, W2 — сервера. Если сервер не понимает имени элемента данных, он отправляет отрицательное подтверждение сообщением WM_DDE_ACK, подставляя 0 вместо N:

```
LPARAM lParam = PackDDElParam(WM_DDE_ACK, N, atom);
PostMessage(W1, WM_DDE_ACK, (WPARAM)W2, lParam);
```

Если сервер понимает запрос, то резервирует память hGlobal размером DDEDATA плюс размер данных, блокирует ее, записывает в память флаги, формат, данные, и отправляет клиенту сообщение WM_DDE_DATA:

```
LPARAM lParam = PackDDElParam(WM_DDE_DATA, (UINT)hGlobal, atom));
PostMessage(W1, WM_DDE_DATA, (WPARAM)W2, lParam);
```

Клиент, получив сообщение WM_DDE_DATA, извлекает параметры:

```
UnpackDDElParam(WM_DDE_DATA, lParam, (PUINT)&hGlobal, (PUINT)&atom);
```

Если не удалось блокировать данные или данные пришли в ином формате, клиент посылает отрицательное подтверждение. Если данные успешно извлечены и требуется подтверждение, клиент посылает положительное подтверждение, записывая вместо N значение 0x8000.

Для установления постоянной связи клиент посылает серверу сообщение WM_DDE_ADVISE, записывая в структуру DDEADVISE параметры: формат данных cfFormat, признак подтверждения fAckReq, и признак типа связи fDeferUpd, равный 0 для горячей связи, или 1 для теплой. Сервер отвечает положительным или отрицательным подтверждением.

Если установлена горячая связь, данные будут приходить обычным образом с сообщением WM_DDE_DATA. При теплой связи это же сообщение будет иметь указатель hGlobal, равный нулю, и клиент запрашивает данные помощи обычным образом, сообщением WM_DDE_REQUSET.

Клиент посылает данные серверу так же, как сервер клиенту, только используя сообщение WM_DDE_POKE и структуру DDEPOKE.

Команды серверу имеют вид [команда(аргумент, аргумент)]. Клиент выделяет память hGlobal, блокирует ее, список команд копируется в эту память, и посылается сообщение WM_DDE_EXECUTE:

```
lParam = PackDDElParam(WM_DDE_EXECUTE, 0, (UINT)hGlobal);
PostMessage(W2, WM_DDE_EXECUTE, (WPARAM)W1, lParam));
```

В обязанности участников входит также освобождение разделяемой памяти hGlobal, освобождение lParam, уничтожение атомов. Примеры кода можно найти на сайте Microsoft по запросу «msdn.docs dynamic data exchange».

8.4. Почтовые ящики

Почтовый ящик — это разделяемая между процессами область памяти, в которую можно записывать сообщения, и затем считывать их оттуда. Почтовый ящик можно рассматривать как дисковый файл, с тем отличием, что он находится в памяти и существует временно. Максимальный размер ящика составляет 64 Кбайт.

Процессы, работающие с почтовым ящиком, делятся на серверные и клиентские. Серверный процесс создает почтовый ящик, после чего может извлекать из него сведения о сообщениях, считывать сообщения, и при определенных обстоятельствах сам записывать сообщения в ящик. Считанное сообщение удаляется из ящика. Клиентские процессы могут только записывать сообщения.

Сервер создает почтовый ящик функцией CreateMailslot:

```
HANDLE CreateMailslot(
    LPCTSTR lpName,             // имя почтового ящика
    DWORD nMaxMessageSize,      // максимальный размер сообщения
    DWORD lReadTimeout,         // тайм-аут чтения
    LPSECURITY_ATTRIBUTES lpSecurityAttributes // наследование
);
```

Имя почтового ящика имеет одну из следующих форм:

```
\\.\mailslot\name1
\\.\mailslot\name3\name2\name1
```

Здесь name1 — имя ящика, другие имена — путь. Максимальная длина имени 256, имя не зависит от регистра.

В программе имя может быть задано следующим образом:

```
LPCSTR TestSlot = TEXT("\\\\.\\mailslot\\testslot");
```

Данная форма имени используется при создании почтового ящика сервером, и фактически ящик находится на компьютере сервера. Клиент может использовать как эту форму, так и любую из следующих:

```
\\computer-name\mailslot\[путь]имя
\\domain-name\mailslot\[путь]имя
\\*\mailslot\[путь]имя
```

Первая форма указывает имя компьютера, который содержит ящик. Вторая форма указывает все компьютеры заданного домена, а третья — все компьютеры первичного системного домена, в которых есть почтовый ящик, имеющий заданное имя и путь. Таким образом, почтовый ящик доступен на удаленных компьютерах сети. Если размер сообщения менее 425 байт, сообщение отправляется по сети при помощи протокола UDP, иначе при помощи протокола SMB, при этом нельзя записать сообщение в несколько ящиков. Кроме того, в Windows нельзя использовать размер сообщения в 425 и 426 байт.

Максимальный размер сообщения может быть задан нулевым, это означает отсутствие ограничения. Тайм-аут задает время в миллисекундах, которое операция чтения будет ждать, пока появится сообщение. Если задать нулевое значение, функция чтения завершается немедленно, если ящик пуст. Если задать значение MAILSLOT_WAIT_FOREVER, функция чтения будет ожидать вечно. Атрибуты безопасности, кроме всего прочего, позволяют наследовать дескриптор (серверными процессами).

Запись и чтение выполняют обычные функции, WriteFile, WriteFileEx, ReadFile, ReadFileEx. Эти операции могут быть асинхронными, с использованием структуры OVERLAPPED. Асинхронная операция предотвращает зависание оконного приложения, если функция долго ждет.

Для записи клиент открывает почтовый ящик как файл обычным образом, при помощи функции CreateFile. Сервер может сделать то же самое, и тогда у него будет два дескриптора, для чтения, и для записи.

Для почтового ящика определены еще только две функции.

Функция SetMailslotInfo задает тайм-аут чтения:

```
BOOL SetMailslotInfo(
  HANDLE hMailslot,    // дескриптор
  DWORD lReadTimeout   // тайм-аут
);
```

Функция GetMailslotInfo получает информацию о сообщениях:

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,          // дескриптор
    LPDWORD lpMaxMessageSize,  // максимальный размер сообщения
    LPDWORD lpNextSize,        // размер следующего сообщения
    LPDWORD lpMessageCount,    // число сообщений
    LPDWORD lpReadTimeout      // тайм-аут
);
```

Вместо второго и последнего параметра можно подставлять NULL.

8.5. Каналы

Канал можно представить как трубку, через которую передаются сообщения, они потому так и называются (pipe — труба). Существуют анонимные и поименованные каналы. Анонимные каналы используются для передачи данных между родительским и дочерним процессами на одном компьютере. Именованные каналы можно использовать для передачи сообщений как на одном компьютере, так и по сети.

Здесь также различают серверный и клиентский процессы, но это соотношение не постоянное, один и тот же процесс может выступать и как сервер и как клиент, но в разные моменты времени. Процесс серверный, если он создает канал Процесс клиентский, если он подключается к открытому каналу. Мы рассматриваем только именованные каналы.

Именованный канал создает функция CreateNamedPipe:

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,         // имя канала
    DWORD dwOpenMode,       // режим открытия
    DWORD dwPipeMode,       // режим канала
    DWORD nMaxInstances,    // максимальное число экземпляров
    DWORD nOutBufferSize,   // размер выходного буфера
    DWORD nInBufferSize,    // размер входного буфера
    DWORD nDefaultTimeOut, // тайм-аут
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Имя канала задается следующим образом:

```
\\.\pipe\pipename               // для сервера или клиента
\\server-name\pipe\pipename     // для клиента
```

Здесь server-name — имя сервера, pipename — имя канала. Полная длина не более 256, имя не зависит от регистра. Путей в имени нет.

Режим открытия канала задают следующие константы:

PIPE_ACCESS_INBOUND — клиент записывает, сервер читает;

PIPE_ACCESS_OUTBOUND — сервер записывает, клиент читает;

PIPE_ACCESS_DUPLEX — сервер и клиент записывают и читают.

К одной из этих констант можно добавить следующие:

FILE_FLAG_WRITE_THROUGH — режим write-through, когда функция записи в канал не возвращается до тех пор, пока данные не поступят к получателю. Имеет значение, когда канал ориентирован на поток байт и передача происходит по сети.

FILE_FLAG_OVERLAPPED — используется асинхронная операция.

Режим канала задают следующие три группы констант.

Тип канала:

PIPE_TYPE_BYTE — канал байт-ориентированный (по умолчанию).

PIPE_TYPE_MESSAGE — канал ориентирован на передачу сообщений.

Режим чтения:

PIPE_READMODE_BYTE — байт-ориентированное (по умолчанию).

PIPE_READMODE_MESSAGE — ориентированное на сообщения.

Если канал ориентирован на передачу потока байт, то режим чтения может быть только байт-ориентированным.

Режим ожидания:

PIPE_WAIT — блокирующий режим, функция ждет подключения.

PIPE_NOWAIT — функция возвращается немедленно.

Максимальное число экземпляров задает количество каналов с одним именем. Все экземпляры должны иметь одинаковые режим открытия, тип канала, максимальное число экземпляров и тайм-аут.

Размеры буферов, резервируемых для входа и выхода, округляются, или определяются системой, если задать их равными нулю.

Тайм-аут имеет значение в случае, если функция WaitNamedPipe задает параметр NMPWAIT_USE_DEFAULT_WAIT.

В случае неудачи функция CreateNamedPipe возвращает значение INVALID_HANDLE_VALUE. В случае удачи сервер ожидает подключения к каналу при помощи функции ConnectNamedPipe:

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,          // дескриптор канала
    LPOVERLAPPED lpOverlapped   // структура overlapped
);
```

Структура OVERLAPPED используется, если операция асинхронная. В этом случае эта структура должна содержать дескриптор события с ручным сбросом (manual-reset), его создает CreateEvent(0, 0, 0, 0). Функция возвращает нуль в случае неудачи, однако если GetLastError при этом возвращает ERROR_PIPE_CONNECTED, то к каналу успел подключиться клиент, и это нормальное установление соединения. Функция применяется либо к новому экземпляру канала, либо к экземпляру, который завершил сеанс, и отключен функцией DisconnectNamedPipe.

Клиент подключается к каналу при помощи разных функций. Если используется функция CreateFile и нет ни одного свободного экземпляра канала, CreateFile возвращает нуль, а GetLastError — ERROR_PIPE_BUSY. В этом случае клиент использует функцию WaitNamedPipe, которая ждет освобождения какого-нибудь экземпляра. Функция CreateFile завершается неудачно также в случае, когда не совпадают режимы доступа. Для канала типа INBOUND режим доступа может быть только GENERIC_WRITE, для канала типа OUTBOUND режим доступа только GENERIC_READ, и для канала типа DUPLEX режим доступа любой.

Функция CallNamedPipe используется для подключения к каналу, ориентированному на сообщения, она ожидает освобождения канала, если все экземпляры заняты, затем записывает в канал, читает его, после чего закрывает канал (см. также функцию TransactNamedPipe):

```
BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName,    // имя канала
    LPVOID lpInBuffer,          // буфер записи
    DWORD nInBufferSize,        // размер буфера записи
    LPVOID lpOutBuffer,         // буфер чтения
    DWORD nOutBufferSize,       // размер буфера чтения
    LPDWORD lpBytesRead,        // число прочитанных байт
    DWORD nTimeOut              // тайм-аут ожидания канала
);
```

Последний параметр может быть задан также константами:

NMPWAIT_NOWAIT — не ждать, если занят, завершить функцию.

NMPWAIT_WAIT_FOREVER — ждать, пока не отключат электричество.

NMPWAIT_USE_DEFAULT_WAIT — ждать, как задано в CreateNamedPipe.

9. Сервисы NT

Сервис NT — это обычно консольное приложение, работающее в фоновом режиме и построенное по определенным правилам. Заправляет сервисами программа «Менеджер управления сервисами» SCM (Service Control Manager), некоторые функции которой доступны через консоль Services или через команду sc. SCM ведет одну или несколько баз данных, в которых записывается конфигурационная информация сервисов.

Сервис стартует автоматически при старте системы, вручную при помощи консоли Services, или программно при помощи функций сервисов. Сервисы функционируют независимо от наличия пользователей в системе, используя для этого учетную запись локальной машины. Для связи приложений с сервисами используют именованные каналы.

9.1. Структура сервиса

Сервис обязан содержать следующие функции:
main — точка старта консольного приложения;
service_main — условное название точки входа в сервис;
control_handler — условное название функции управления сервисом.

9.1.1. Основная функция

Примерный вид основной функции следующий:

```
// прототип функции сервиса
VOID service_main(DWORD argc, LPSTR *argv);
void main(возможные параметры) {
    SERVICE_TABLE_ENTRY DispatchTable[] =  {
        { "MyService", service_main },
        { 0, 0 }
    };
    if (!StartServiceCtrlDispatcher(DispatchTable)) {
        // ошибка инициализации сервиса
    }
}
```

Как видим, функция содержит таблицу точек входа, поскольку один процесс может содержать несколько сервисов. Эти точки входа регистрирует функция StartServiceCtrlDispatcher. В связи с этим различают два типа сервисов: SERVICE_WIN32_OWN_PROCESS — программа-сервис, содержащий код только одного сервиса; и SERVICE_WIN32_SHARE_PROCESS — программа-сервис, содержащий код нескольких сервисов. Когда сервис создается при помощи функции CreateService, одна из этих констант указывает, какого типа сервис. Кроме просто сервисов, существуют еще сервисы-драйверы и сервисы-драйверы файловой системы.

Если функция StartServiceCtrlDispatcher успешно выполняется, тогда первичный поток находится в ней *до завершения работы сервиса*.

9.1.2. Функция сервиса

Функция сервиса с условным названием service_main является точкой входа в сервис. Ее структура проста: инициализация необходимых структур данных, затем обслуживание, обычно это бесконечный цикл ожидания подключения по каналу, и завершение (очистка).

Вид функции сервиса в общих чертах следующий:

```
SERVICE_STATUS sStatus;
SERVICE_STATUS_HANDLER hStatus;
void WINAPI service_main(DWORD dwArgc, LPTSTR *lpszArgv) {
    // регистрируем функцию управления
    hStatus = RegisterServiceCtrlHandler(szSERVNAME, service_ctrl);
    if (!hStatus) { // ошибка регистрации
        return;
    }
    sStatus.dwCurrentState = SERVICE_START_PENDING;
    // заполняем другие поля структуры SERVICE_STATUS
    // уведомляем SCM о состоянии инициализации
    SetServiceStatus(hStatus, sStatus);
    // инициализация, например, при помощи вспомогательной функции
    if (!Initialization()) { // ошибка инициализации
        sStatus.dwCurrentState = SERVICE_STOPPED;
        // уведомляем SCM об остановке
        SetServiceStatus(hStatus, sStatus);
        return;
    }
    sStatus.dwCurrentState = SERVICE_RUNNING;
    // уведомляем SCM о рабочем состоянии
    SetServiceStatus(hStatus, sStatus);
    // основной цикл
    while (1) {
        // выполняем основную работу
    }
    sStatus.dwCurrentState = SERVICE_STOP_PENDING;
    // уведомляем SCM о состоянии завершения
    SetServiceStatus(hStatus, sStatus);
    // выполняем очистку
    sStatus.dwCurrentState = SERVICE_STOPPED;
    // уведомляем SCM об остановке
    SetServiceStatus(hStatus, sStatus);
}
```

Прежде всего нужно зарегистрировать обратно-вызываемую функцию управления сервисом с условным названием service_ctrl, и получить дескриптор hStatus, который используется для подтверждения состояния сервиса. Когда SCM стартует сервис, он ожидает, что сервис не более чем через 1 секунду будет уведомлять SCM о своем состоянии. Функция SetServiceStatus использует для этого структуру SERVICE_STATUS. Поля этой структуры описывают разные аспекты состояния, например, контрольные точки, поле dwCurrentState описывает состояние.

Состояния сервиса задают следующими константами:

SERVICE_START_PENDING — инициализация или завершение;

SERVICE_CONTINUE_PENDING — инициализация или завершение;

SERVICE_RUNNING — вошел в рабочий режим, работает;

SERVICE_STOP_PENDING — выполняет очистку (завершает);

SERVICE_STOPPED — остановлен;

SERVICE_PAUSED — приостановлен.

Во время инициализации и завершения уведомления посылаются либо каждую секунду, либо через интервал времени, задаваемый в поле dwWaitHint. Если вовремя не уведомить SCM, он решит, что сервис завис и выдаст ошибку 1053. Как только сервис начнет работать, уведомления перестают быть нужными до остановки или приостановки.

9.1.3. Функция управления сервисом

Примерный вид функции управления сервисом следующий:

```
VOID WINAPI service_ctrl(DWORD dwCtrlCode) {
    switch (dwCtrlCode) {
        case SERVICE_CONTROL_PAUSE:  // приостановить
        sStatus.dwCurrentState = SERVICE_PAUSED;
        // соответствующие действия
        break;
    case SERVICE_CONTROL_CONTINUE:         // возобновить работу
        sStatus.dwCurrentState = SERVICE_RUNNING;
        // соответствующие действия
        break;
    case SERVICE_CONTROL_STOP:        // остановить *****
        sStatus.dwCurrentState = SERVICE_STOP_PENDING;
        SetServiceStatus(hStatus, sStatus);
        ServiceStop();
        return;
    case SERVICE_CONTROL_INTERROGATE:// обновить состояние *****
        break;
    default:; // недопустимый код управления
    }
    SetServiceStatus(hStatus, sStatus);
}
```

Во всех случаях сервис обязан уведомить о текущем состоянии. SCM посылает команду обновления только для того, чтобы узнать это состояние. Сервис обязан отвечать только на две команды — остановки и обновления (помечены звездочками), а принимаемые сервисом другие команды передаются SCM через поле dwControlsAccepted.

Разберем, как сервис останавливается. Работая, сервис находится в бесконечном цикле обслуживания, ожидания подключения потенциального клиента. Сервис использует режим асинхронного ввода-вывода, так как иначе он просто зависнет в ожидании подключения.

В этом режиме функции ConnectNamedPipe, WriteFile и ReadFile немедленно возвращаются, устанавливая ошибку 997 ERROR_IO_PENDING (выполняется операция ввода-вывода). Завершение операции ожидает одна из Wait-функций, из которой выводит объект события структуры OVERLAPPED. Wait-функция может ожидать несколько событий, поэтому одно из них назначают для остановки. Функция ServiceStop переводит это событие в сигнальное состояние. Wait-функция завершается, сервис анализирует причину, и если причина в событии остановки, завершает цикл (выполняет break).

9.2. Управление сервисами

Хотя сервис представляется обычным консольным приложением, его нельзя просто взять и запустить, ничего не произойдет, потому что функция StartServiceCtrlDispatcher тут же завершится и приложение тоже. Сервис должен быть установлен в базу данных SCM и зарегистрирован в реестре в ключе HKLM\SYSTEM\CurrentControlSet\Services, после чего его работой управляет SCM.

Программно это делается следующим образом:

```
hSCDatabase = OpenSCManager(0, 0, SC_MANAGER_ALL_ACCESS);
if (!hSCDatabase) {
    /* ошибка */
    return;
}
hService = CreateService(
    hSCDatabase,                // база данных SCM
    SZ_SERVICE_NAME,            // имя сервиса программное
    SZ_SERVICE_DISPLAYNAME,     // отображаемое в Services имя
    SERVICE_ALL_ACCESS,         // доступ
    SERVICE_WIN32_OWN_PROCESS,  // тип сервиса
    SERVICE_DEMAND_START,       // тип старта
    SERVICE_ERROR_NORMAL,       // управление ошибками
    szPath,                     // путь к файлу сервиса
    0,                          // нет порядка загрузки групп
    0,                          // нет тега
    "",                         // нет зависимостей
    0,                          // счет LocalSystem account
    0);                         // нет пароля
if (hService) {
    CloseServiceHandle(hService);
} else {
    /* ошибка */
    return;
}
CloseServiceHandle(hSCDatabase);
```

Первый параметр функции OpenSCManager — это имя компьютера, второй — имя базы данных. Функция CreateService устанавливает сервис в заданную базу данных.

В консоли эти действия выполняет команда:

```
sc create имя-сервиса binPath= спецификация-файла-сервиса опции
```

Далее сервис можно запустить, используя консоль Services, либо программно при помощи функции StartService, либо введя команду

```
sc start имя-сервиса
```

Остановить сервис можно в консоли Services, программно при помощи функции ControlService, или командой

```
sc stop имя-сервиса
```

Сервис удаляет из базы данных функция DeleteService или команда

```
sc delete имя-сервиса
```

В русскоязычной Windows 10 Services называется «Службы».

9.3. Сообщения сервиса

Сервис может показывать сообщения функцией MessageBox, но при этом нужно учесть следующее:

1) должен быть включен флаг MB_SERVICE_NOTIFICATION;

2) в Windows 10 эти сообщения попадают в журнал System под названием Application Popup.

Любое приложение может записывать свои сообщения в журнал Application. Сервис относится к классу Application, поэтому он также может записывать сообщения в этот журнал (иначе — лог-файл).

Сообщение записывает функция ReportEvent:

```
BOOL ReportEvent(
    HANDLE hEventLog,     // дескриптор лог-файла
    WORD wType,           // тип события
    WORD wCategory,       // категория события
    DWORD dwEventID,      // идентификатор сообщения
    PSID lpUserSid,       // SID пользователя
    WORD wNumStrings,     // число передаваемых строк
    DWORD dwDataSize,     // размер двоичных данных
    LPCTSTR *lpStrings,   // массив передаваемых строк
    LPVOID lpRawData      // двоичные данные
);
```

Дескриптор лог-файла возвращает функция RegisterEventSource:

```
HANDLE RegisterEventSource(
    LPCTSTR lpUNCServerName,  // имя сервера (машины)
    LPCTSTR lpSourceName      // source (имя-сервиса)
);
```

Источник сообщений должен быть записан в реестре в ключе:

```
HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\имя-сервиса
```

Ключ имя-сервиса должен иметь параметр EventMessageFile, в который записывается путь к источнику сообщений, в качестве которого выступает либо файл сервиса, либо специально созданная DLL.

Для формирования источника сообщений создается файл сообщений (message file) с расширением .mc, описывающий сообщения. Команда mc (message compiler, компилятор ошибок) компилирует этот файл с получением файла ресурсов .rc, заголовочного файла с описанием ошибок, полностью аналогичному файлу winerror.h, и двоичного файла ресурса. Далее компилятор сообщений может сгенерировать DLL.

В файл типа .mc сообщение записывается в следующем виде:

```
MessageId=0x1
Severity=Informational
Facility=Runtime
SymbolicName=MSG_SERVICE_ENTER
Language=English
Reverser has entered service_main
.
```

Параметры: MessageId — номер ошибки, Severety — серьезность ошибки (Success, Informational, Warning или Error), Facility — подсистема, SymbolicName — константа сообщения, Language — язык сообщения. После языка записывается само сообщение и строка с точкой. После строчки с точкой идет либо следующая запись, либо параметр Language и сообщение на указанном языке, либо конец файла.

Пусть файл сообщений называется messages.mc.

Тогда следующая команда компилирует его:

```
mc messages.mc
```

Следующая команда создает файл ресурсов:

```
rc -r -fo messages.res messages.rc
```

Следующая команда создает DLL-файл только с ресурсами:

```
link -dll -noentry -out:messages.dll messages.res
```

При вызове функции ReportEvent указывается тип события, константа EVENTLOG_SUCCESS, EVENTLOG_ERROR_TYPE, EVENTLOG_WARNING_TYPE или EVENTLOG_INFORMATION_TYPE, идентификатор события, заданный в файле .mc как параметр SymbolicName и доступный в программе через заголовочный файл, генерируемый компилятором сообщений. Дополнительно можно задавать категорию события, но для этого нужен более сложный файл сообщений.

Дополнительные сведения о сервисах и примеры кода можно найти на официальном сайте Microsoft docs.microsoft.com, или введя в обозревателе запрос «msdn.docs services». Для поиска сведений о компиляторе сообщений введите запрос «msdn.docs message compiler».

Источники информации

1. MSDN, October 2001.

2. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ. — 4-е изд. — СПб.: Питер; Издательско-торговый дом «Русская Редакция», 2001. — 753 с.: ил.

3. Кокорева О. И. Реестр Microsoft® Windows Server 2003/ — СПб.: БХВ-Петербург, 2003. — 640 с.: ил.

Владимир Вадимович Пономарев
Системное программирование
Учебно-методическое пособие

Наиболее актуальную версию пособия см. revol.ponocom.ru

Отпечатано с готового оригинал-макета
Издательство ОТИ НИЯУ МИФИ, 2019
Тираж 11 экз.