Н.В. МАЙСТРЕНКО, А.В. МАЙСТРЕНКО

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ САПР. ОПЕРАЦИОННЫЕ СИСТЕМЫ

♦ **ИЗДАТЕЛЬСТВО ТГТУ** ♦

УДК 004.451
ББК Ж2-5-05я73-5
М149

Р е ц е н з е н т ы:

Заведующий кафедрой компьютерного и математического моделирования
ТГУ им. Г.Р. Державина
доктор технических наук, профессор
А.А. Арзамасцев

Заведующий кафедрой "Информационные технологии и
защита информации" ТГТУ
доктор технических наук, профессор
В.Н. Шамкин

Майстренко, Н.В.

М149 Программное обеспечение САПР. Операционные системы : учебное пособие / Н.В. Майстренко, А.В. Майстренко. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2007. – 76 с. – 100 экз. – ISBN 5-8265-0596-6 (978-5-8265-0596-0).

Рассмотрены теоретические аспекты разработки системного программного обеспечения, а также практические приемы и навыки в области разработки, отладки и применения современных операционных систем и системного программного обеспечения.

Предназначено для студентов специальности 230104 "Системы автоматизированного проектирования" всех форм обучения, может быть полезным и для студентов, бакалавров и магистров других специальностей и направлений, аспирантов и преподавателей, знакомящихся с работой операционных систем.

УДК 004.451
ББК Ж2-5-05я73-5

ISBN 5-8265-0596-6
(978-5-8265-05969-0)

Министерство образования и науки Российской Федерации

ГОУ ВПО "Тамбовский государственный технический университет"

Н.В. Майстренко, А.В. Майстренко

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ САПР. ОПЕРАЦИОННЫЕ СИСТЕМЫ

Утверждено Ученым советом университета
в качестве учебного пособия для студентов 2 курса
всех форм обучения по специальности 230104

Тамбов
Издательство ТГТУ
2007

Учебное издание

МАЙСТРЕНКО Наталья Владимировна,
МАЙСТРЕНКО Александр Владимирович

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ САПР.
ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебное пособие

Редактор З.Г. Чернова
Инженер по компьютерному макетированию М.Н. Рыжкова

Подписано в печать 28.05.2007.
Формат 60 × 84/16. 4,42 усл. печ. л. Тираж 100 экз. Заказ № 350.

Издательско-полиграфический центр
Тамбовского государственного технического университета
392000, Тамбов, Советская, 106, к. 14

ВВЕДЕНИЕ

Эффективность применения средств вычислительной техники (СВТ) определяется техническим совершенством аппаратной части электронных вычислительных машин (ЭВМ) и вычислительных систем (ВС), качеством программного обеспечения (ПО) и квалификацией персонала, эксплуатирующего СВТ.

Все программное обеспечение ЭВМ можно разделить на две большие группы. Первая группа ПО – *общее программное обеспечение,* объединяет в себе программы, описания и инструкции, предназначенные для автоматизации трудоемких технологических этапов разработки алгоритмов и программ (инструментальные средства разработки ПО), сервисные программы, предназначенные для обслуживания ЭВМ и ВС (утилиты), и программные комплексы для организации и контроля вычислительного процесса в ВС во время ее функционирования (операционные системы – ОС). Для того чтобы ориентировать вычислительную систему на решение задач определенного типа, к общему программному обеспечению добавляется вторая группа ПО – *специальное программное обеспечение* (называемое также *прикладным программным обеспечением* или *пакетами прикладных программ*), которое позволяет эффективно использовать ВС в зависимости от конкретной сферы ее применения.

Целью настоящего учебного пособия является рассмотрение прак-тических приемов и навыков в области разработки, отладки и применения современных операционных систем и системного программного обеспечения.

В рамках изучения дисциплины студентам предлагается выполнить шесть лабораторных работ, позволяющих изучить архитектуру и внутреннее строение операционной системы на примере операционных систем семейства Windows.

При выполнении каждой лабораторной работы студенту необходимо изучить теоретический материал к ней, выполнить индивидуальное задание, выданное преподавателем, оформить и защитить отчет по лабораторной работе.

Каждый отчет по лабораторной работе должен содержать название лабораторной работы и ее цель, последовательность этапов выполнения лабораторной работы, выводы и список использованной литературы. Кроме этого, к отчету обязательно прилагаются листинги разработанных программ и результаты их работы.

Лабораторная работа 1

ФУНКЦИИ ПОЛУЧЕНИЯ СИСТЕМНОЙ ИНФОРМАЦИИ

Цель работы: получение практических навыков по программированию в Win32 API с использованием аппаратных и системных функций.

ОСНОВНЫЕ ПОЛОЖЕНИЯ

В настоящее время Microsoft предлагает две линейки операционных систем: Windows 9x (Windows 95, 98, ME) и Windows NT (Windows NT, 2000, XP, Vista). Интерфейс прикладного программирования (Application Programming Interface Win32 – Win32 API) – это программный интерфейс, который используется для управления этими операционными системами. Win32 API состоит из набора функций и подпрограмм, предоставляющих программный доступ к возможностям операционной системы, т.е. программные интерфейсы приложений представляют собой наборы функций (в этот обобщенный термин включаются и подпрограммы), которые обеспечивают сервисы данного приложения. Win32 API содержит более 3000 функции для реализации всех видов сервисов операционной системы.

API-функции Windows входят в состав динамически подключаемых библиотек. Динамически подключаемая библиотека (Dynamic Link Library – DLL) является исполняемым файлом, который содержит несколько экспортируемых функции (exportable functions), т.е. функций, к которым могут обращаться другие исполняемые приложения (EXE или DLL). Файлы DLL намного проще файлов EXE, например, в них нет кода, который управлял бы графическим интерфейсом или обрабатывал сообщения Windows.

Для размещения API-функций Windows использует несколько DLL. В действительности большая часть функций Win32 API содержится в трех DLL:

- KERNEL32.DLL – содержит около 700 функций, которые предназначены для управления памятью, процессами и потоками;
- USER32.DLL – предоставляет порядка 600 функций для управления пользовательским интерфейсом, например, созданием окон и передачей сообщений;
- GDI.DLL – экспортирует около 400 функций для рисования графических образов, отображения текста и работы со шрифтами.

Кроме названных библиотек Windows также содержит несколько других DLL более узкой специализации:

- COMDLG32.DLL – открывает доступ почти к 20 функциям управления стандартными диалоговыми окнами Windows;
- LZ32.DLL – хранит примерно 12 функций архивирования и разархивирования файлов;
- ADVAPI32.DLL – экспортирует около 400 функций, связанных с защитой объектов и работой с реестром;
- WINMM.DLL – содержит около 200 функций, относящихся к мультимедиа.

Основные Win32 API-функции получения системной информации:

GetComputerName	GetSystemMetrics	GetWindowsDirectoty
GetKeyboardType	GetTempPath	SetComputerName
GetSysColor	GetUserName	SetSysColors
CecSystemDirectory	GetVersion	SystemParametersInfo
GetSystemlnfo	GetVersioriEx	GetUserName

Имя компьютера

Функция GetComputerName используется для получения текущего имени компьютера. Связанная с ней SetComputerName используется для присвоения имени компьютеру:

```
BOOL GetComputerName(
LPTSTR IpBuffer,                    // Адрес буфера имени.
LPDWORD nSize                       // Размер буфера имени.
);
```

В соответствии с документацией, выполнение функции GetComputer Name в Windows 9x завершится неудачей, если размер буфера входных данных меньше, чем величина константы MAX_COMPUTERNAME_ LENGTH + 1.

Пути к системным каталогам Windows

Функции GetWindowsDirectory, GetSystemDirectory и GetTempPath находят путь к каталогу, к системному каталогу и к каталогу временных файлов Windows. Например, функция GetSystemDirectory определена как:

```
UINT GetSystemDirectory(
LPTSTR IpBuffer,                    // Адрес буфера системного каталога.
UINT nSize                          // Размер буфера каталога.
);

UINT GetWindowsDirectory(
LPTSTR IpBuffer,                    // Адрес буфера каталога Windows.
UINT nSize                          // Размер буфера каталога.
);
DWORD GetTempPath(
DWORD nBufferLength,                // Размер буфера в символах.
LPTSTR IpBuffer                     // Указатель на буфер пути к каталогу
                                    // временных файлов.
);
```

Версия операционной системы

Функция GetVersionEx возвращает информацию о версии операционной системы Windows и может использоваться для определения рабочей системы – Windows 95, Windows 98 или Windows NT. Она объявляется как

```
BOOL GetVersionEx(
LPOSVERSIONINFO                     // Указатель на структуру
IpVersionlnformation                // с информацией о версии.
);
```

где IpVersionlnformation – указатель на структуру OSVERSIONINFO, которая определена следующим образом:

```
typedef struct _OSVERSIONINFO {
DWORD dwOSVersionlnfoSise;
DWORD dwMajorVersion;
DWORD dwMinorVersion;
DWORD dwBuildNumber;
DWORD dwPlatformId;
TCHAR szCSDVersion[ 128 ];
} OSVERSIONINFO;
```

dwOSVersionlnfoSize задает размер структуры OSVERSIONINFO в байтах, что для структур является общим требованием. Так как DWORD –четырехбайтовое беззнаковое типа long и поскольку Delphi и VB преобразуют строку из 128 символов в массив символов ANSI из 128 байт, общий размер структуры составляет 4x5 + 128 = 148 байт. Это значение возвращает функция Len для VB и SizeOf для Delphi.

dwMajorVersion указывает номер основной версии операционной системы. Например, для Windows NT версии 3.51 номер основной версии – 3. Для Windows NT 4.0 и Windows 9x номер основной версии – 4.

dwMinorVersion указывает дополнительный номер версии операционной системы. Например, Для Windows NT версии 3.51 дополнительный номер версии – 51. Для Windows NT 4.0 и Windows 95 дополнительный номер версии – 0. Для Windows 98 дополнительный номер версии – 10.

dwBuildNumber указывает номер сборки операционной системы для Windows NT. Для Windows 9x два младших байта содержат номер сборки операционной системы, а два старших байта – номер основной версии и дополнительный номер версии.

dwPiatformId идентифицирует платформу операционной системы, может иметь одно из следующих значений:

VER_PLATFORM_WIN32s (= 0) // Win32s, работающая на
// Windows.
VER_PLATFORM_WIN32_WINDOWS (= 1). // Win32, работающая на
// Windows 95 или
// Windows 98.
VER_PLATFORM_WIN32_NT (= 2) // Win32, работающая на
// Windows NT.

szCSDVersion. В Windows NT содержит строку, завершающеюся нулевым символом, например «Service Pack3», которая указывает самую последнюю версию установленного в системе служебного пакета программ (service pack). Строка будет пустой, если служебный пакет не установлен. В Windows 95 включает строку с завершающим нулевым символом, в которой может быть произвольная дополнительная информация об операционной системе.

Системные метрики

Функция GetSystemMetrics получает информацию о метриках (системе единиц измерения) объектов операционной системы:

```
Int GetSystemMetrics(
int nIndex                    // Системная метрика или
                              // установки конфигурации.
);
```

Параметр nIndex принимает значение одной из 84 возможных констант. Функция возвращает запрошенные единицы измерения (в общем случае в пикселах или в безразмерных единицах).

Чтобы дать общее представление о типе возвращаемой информации, здесь приведены образцы некоторых констант для этой функции. Единицы измерения высоты и ширины приведены в пикселах:

```
SM_CMOUSEBUTTOMS = 43         // Количество клавиш мыши.
SM_MOUSEWHEELPRESENT = 75     // Истина (True), если мышь
                              // имеет колесо прокрутки
                              // (Только Win NT 4 или Win 98).
SM  SWAPBUTTON = 23           // Истина (True), если клавиши
                              // мыши можно поменять
                              // местами (мышь для левши).
SM_CXBORDER = 5               // Ширина и высота рамки окна.
SM_CYBORDER = 6
SM_CXSCREEN = 0               // Ширина и высота экрана.
SM_CYSCREEN = 1
SM_CXFULLSCREEN = 16          // Ширина и высота области
SM_CYFULLSCREEN = 17          // приложения в полноэкранном
                              // режиме.
SM_CXHTHUMB = 10              // Ширина прямоугольного курсора
                              // в горизонтальной полосе
                              // прокрутки.
SM_CXICONS PACING = 38        // Размеры ячейки сетки для
SM_CYICONSPACING = 39         // значка в режиме просмотра с
                              // крупными значками.
SM_CYCAPTION =4               // Высота стандартной области
                              // заголовка.
```

Системные параметры

Функция SystemParamterslnfo – это мощная функция, предназначенная для получения или установки всех системных параметров. Она может также в процессе установки параметра обновлять пользовательские профили. Ее декларация:

```
BOOL SystemParametersInfo (   // Запрашиваемый или
UINT uiAction,                // устанавливаемый системный
                              // параметр.
UINT uiParam,                 // Зависит от принятого системного
                              // параметра.
PVOID pvParam,                // Зависит от принятого системного
                              // параметра.
UINT fWinIni                  // Флаг обновления пользовательского
                              // профиля.
};
```

Эта функция может принимать, по меньшей мере, 90 различных значений uiAction. Некоторые константы uiAction:

SPI_GETACCESSTIMEOUT – используется для определения данных о временных интервалах, относящихся к специальным возможностям Windows;

SPI_GETANIMATION – используется для определения данных об анимации, используемой при сворачивании и восстановлении окон;

SPI_GETBEEP – признак разрешения звуковых сигналов;

SPI_GETBORDER – параметру присваивается коэффициент, управляющий толщиной рамки для изменения размеров окна;

SPI_GETDEFAULTINPUTLANG – параметру присваивается 32-разряд-ный дескриптор раскладки клавиатуры по умолчанию;

SPI_GETDRAGFULLWINDOWS – характеристики перемещения окна мышью;

SPI_GETFASTTASKSWITCH – признак, определяющий быстрое переключение задач;

SPI_GETFILTERKEYS – используется для определения данных о специальных возможностях, относящихся к работе с клавиатурой;

SPI_GETFONTSMOOTHING – режимы сглаживания шрифтов;

SPI_GETGRIDGRANULARITY – гранулярность сетки рабочего стола;

SPI_GETICONMETRICS – используется для определения информации о характеристиках иконок.

Системные цвета

Функции GetSysColor и SetSysColors используются для получения и установки цветов различных элементов системы, таких как кнопки, строки заголовков и т.д. Цветовой палитрой может также управлять пользователь с помощью апплета Display (Экран) на панели Control Panel (Панель управления). Декларация GetSysColor:

```
DWORD GetSysColor (
int nIndex                                  // Элемент экрана.
);
```

где *nIndex* может принимать значение одной из множества символьных констант, например

```
#define COLOR_ACTIVECAPTION 3
```

Возвращаемое значение – это цвет в формате RGB. В частности, каждый цвет занимает один байт в возвращаемом значении типа unsigned long: красный. Цвет – младший байт, зеленый – следующий байт, далее – синий цвет. Самый старший байт равен нулю. Байты цветов представлены в переменной типа long в обратном порядке, поскольку при записи переменной в память байты располагаются от младших к старшим.

Объявление функции SetSysColors:

```
BOOL WINAPI SetSysColors (              // Количество изменяемых
int cElements,                          // элементов.
CONST INT *lpaElements,                 // Адрес массива элементов.
CONST COLORREF *lpaRgbValues            // Адрес массива значений RGB.
);
```

Здесь cElements определяет количество системных элементов, цвет которых требуется изменить; IpaElements – указатель на целочисленный массив, который содержит индексы изменяемых элементов; IpaRgbvalues ссылается на целочисленный массив новых значений цвета в формате RGB.

Функции для работы со временем

Во внутренней работе Windows используется универсальное координированное время UTC (Universal Coordinated Time); также встречается термин GMT, т.е. "среднее время по Гринвичу" (Greenwich Mean Time), поскольку за точку отсчета принят Гринвич, Англия. Преобразования между системным и местным временем в Windows осуществляются при помощи поправок для местного часового пояса, заданного в системе. Функции Win32 позволяют работать как в местном, так и в системном времени и преобразовывать их по мере необходимости. Win32 также включает ряд функций для работы с файловым временем и датой, т.е. временем и датой файлов, хранящимся в файловой системе.

Функции Windows, предназначенные для получения информации о времени, перечислены в табл. 1. Следует учитывать, что во внутреннем представлении системы время изменяется в тактах таймера, продолжительность которых может изменяться в зависимости от используемого процессора и операционной системы. Интервал измерения времени в Win32 обычно занимает 10…15 мс. Длительность такта определяет точность результатов, возвращаемых этими функциями.

1. Основные функции Windows для работы со временем

Функция	Описание
EnumCalendarInfo	Перечисляет календарную информацию, зависящую от локального контекста
EnuinDate Formats	Перечисляет форматы даты, доступные в заданном локальном контексте
EnumTimeFormats	Перечисляет форматы времени, доступные в заданном локальном контексте
GetLocalTime	Получает текущее местное время
GetMessageTime	Возвращает время (в миллисекундах) поступления последнего сообщения s очередь приложения. Время отсчитывается от начала текущего сеанса работы в Windows

GetSystemTime	Получает текущее системное время
GetSystemTimeAdjustment	Определяет, применяется ли в системе периодическая поправка, повышающая точность отсчета системного времени
GetTickCount	Получает продолжительность работы текущего сеанса работы в Windows в миллисекундах
GetTimeFormat	Форматирует время в заданном локальном контексте
GetTimeZonelnformation	Получает информацию о текущем часовом поясе
SetLocalTime	Задает местное время
SetSystemTime	Задает системное время
SetSystemTimeAdjustment	Задает периодическую поправку, применяемую системой для повышения точности отсчета времени
SetTimeZonelnformation	Задает часовой пояс
SystemTimeToTzSpecific-LocalTime	Преобразует системное время в местное

ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Разработать программу, обеспечивающую получение системной информации:

1) имя компьютера, имя пользователя;
2) пути к системным каталогам Windows;
3) версия операционной системы;
4) системные метрики (не менее двух метрик);
5) системные параметры (не менее двух параметров);
6) системные цвета (определить цвет для символьных констант и изменить его на любой другой);
7) функции для работы со временем (не менее двух функций);
8) дополнительные API-функции (не менее двух функций).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое API-функции? Какую информацию можно получить с их помощью?
2. Как можно определить версию операционной системы? Структура какого типа для этого используется?
3. Какую информацию получают функцией GetSystemMetrics?
4. Как можно изменить системные цвета? Как формируется значение цвета?
5. Для чего используются дополнительные API-функции?

Лабораторная работа 2

АРХИТЕКТУРА WINDOWS

Цель работы: изучение архитектуры операционной системы Windows.

ОСНОВНЫЕ ПОЛОЖЕНИЯ

Приложение (application) Windows – это совокупность исполняемых программ и вспомогательных файлов. Например, Microsoft Word представляет собой одно из популярных приложений Windows. Процессом называется исполняемый экземпляр приложения. Заметим, что в большинстве случаев пользователь может запускать несколько экземпляров (копий) одного и того же приложения одновременно. Каждый исполняемый экземпляр – это отдельный процесс со своей собственной областью памяти.

Процессом (process) называется исполняемый экземпляр (running instance) приложения и комплект ресурсов, отводящийся данному исполняемому приложению.

Поток (thread) – это внутренняя составляющая процесса, которой операционная система выделяет процессорное время для выполнения кода. Именно потоки исполняют программный код, а не процессы. Каждый процесс должен иметь как минимум один поток. Конечно, основное назначение потоков – дать процессу возможность поддерживать несколько ветвей управления, т.е. выполнять больше действий одновременно. В многопроцессорной конфигурации (компьютер с несколькими процессорами) Windows NT (но не Windows 9x) может распределять потоки по процессорам, реально обеспечивая параллельную обработку. В однопроцессорной конфигурации процессор должен выделять кванты времени (time slices) каждому исполняемому в данный момент потоку.

Архитектура Windows NT в обобщенном виде представлена на рис. 1. Рассмотрим некоторые из изображенных пунктов.

Режим ядра и пользовательский режим

Микропроцессор Pentium имеет четыре уровня привилегий (privilege levels), известных также как кольца (rings), которые управляют, например, доступом к памяти, возможностью использовать некоторые критичные команды процессора (та-

кие как команды, связанные с защитой) и т.д. Каждый поток выполняется на одном из этих уровней привилегий. Кольцо 0 – наиболее привилегированный уровень, с полным доступом ко всей памяти и ко всем командам процессора. Кольцо 3 – наименее привилегированный уровень.

Рис. 1. Архитектура Windows NT в упрощенном виде

Для обеспечения совместимости с системами на базе процессоров, отличных от тех, что выпускает компания Intel, Windows поддерживает только два уровня привилегий – кольца 0 и 3. Если поток работает в кольце 0, говорят, что он выполняется в режиме ядра (kernel mode). Если поток выполняется в кольце 3, говорят, что он работает в пользовательском режиме (user mode). Низкоуровневый код операционной системы действует в режиме ядра, тогда как пользовательские приложения выполняются, в основном, в пользовательском режиме.

Прикладной поток может переключаться из пользовательского режима в режим ядра при вызове некоторых API-функций, которые требуют более высокого уровня привилегий, например, связанных с доступом к файлам или с выполнением функций, ориентированных на графические операции. В действительности некоторые пользовательские потоки могут работать в режиме ядра даже больше времени, чем в пользовательском режиме.

Но как только завершается выполнение той части кода, которая относится к режиму ядра, пользовательский поток автоматически переключается обратно в пользовательский режим. Такой подход лишает возможности писать код, предназначенный для работы в режиме ядра, программист может только вызывать выполняющиеся в режиме ядра системные функции (system functions). При работе с Windows NT можно определить, когда поток выполняется в пользовательском режиме, а когда – в режиме ядра. Для этого используется утилита Performance Monitor (Системный монитор) из пункта Administrative Tools (Администрирование) меню Start (Пуск).

Драйверы устройств работают в режиме ядра. Это обстоятельство имеет два следствия. Во-первых, в отличие от неправильно выполняющегося приложения неправильно работающий драйвер устройства может нарушить работу всей системы, так как он имеет доступ и ко всему системному коду, и ко всей памяти. Во-вторых, прикладной программист может получить доступ к защищенным ресурсам, написав драйвер псевдоустройства.

Сервисы

Термин сервис (service) имеет в среде Windows множество значений. Некоторые из них, имеющие отношение к рассматриваемой теме:

- сервис API – функция или подпрограмма API, которая реализует некоторое действие (сервис) операционной системы, такое как создание файла или работа с графикой (рисование линий или окружностей). Например, функция API CreateProcess используется в Windows для создания нового процесса;
- системный сервис – недокументированная функция, которая может вызываться из пользовательского режима; часто используется функциями Win32 API для предоставления низкоуровневых сервисов;
- внутренний (internal) сервис – функция или подпрограмма, которая может вызываться только из кода, выполняемого в режиме ядра; относится к низкоуровневой части кода Windows: к исполнительной системе Windows NT, к ядру или к слою абстрагирования от аппаратуры (HAL).

Системные процессы

Системные процессы (system processes) – это особые процессы, обслуживающие операционную систему. В системе Windows постоянно задействованы следующие системные процессы (учтите, что все они, кроме процесса system, выполняются в пользовательском режиме):

- процесс idle, который состоит из одного потока, управляющего временем простоя процессора;
- процесс system – специальный процесс, выполняющийся только в режиме ядра; его потоки называются системными потоками (system threads);
- процесс Session Manager (диспетчер сеансов) – SMSS.EXE;
- подсистема Win32 – CSRSS.EXE;
- процесс регистрации в системе – WinLogon (WINLOGON.EXE).

Процесс Session Manager (SMSS.EXE) – один из первых процессов, создаваемых операционной системой в процессе загрузки. Он выполняет важные функции инициализации – такие как создание переменных окружения системы; задание имен устройств MS DOS, например, LPT1 и COM1; загрузка той части подсистемы Win32, которая относится к режиму ядра; запуск процесса регистрации в системе WinLogon.

Процесс WinLogon управляет входом пользователей в систему и выходом из нее. Вызывается специальной комбинацией клавиш Windows Ctrl+Alt+Delete. WinLogon отвечает за загрузку оболочки Windows (обычно, это Windows Explorer).

Процесс system состоит из системных потоков (system threads), являющихся потоками режима ядра. Windows и многие драйверы устройств создают потоки процесса system для различных целей. Например, диспетчер памяти формирует системные потоки для решения задач управления виртуальной памятью, диспетчер кэша использует системные потоки для управления кэш-памятью, а драйвер гибкого диска – для контроля над гибкими дисками.

Подсистема Win32 является разновидностью подсистемы среды (environment subsystem). Другие подсистемы среды Windows (не показаны на рисунке) включают POSIX и OS/2. POSIX является сокращением термина "переносимая операционная система на базе UNIX" (portable operating system based on UNIX) и реализует ограниченную поддержку операционной системы UNIX.

Назначение подсистемы среды – служить интерфейсом между пользовательскими приложениями и соответствующей частью исполнительной системы Windows. Каждая подсистема имеет свои функциональные возможности на базе единой исполнительной системы Windows. Любой выполняемый файл неразрывно связан с одной из этих подсистем. Подсистема Win32 содержит Win32 API в виде набора DLL – таких как KERNEL32.DLL, GDI32.DLL и USER32.DLL.

В Windows NT Microsoft перенесла часть подсистемы Win32 из пользовательского режима в режим ядра. В частности, драйвер устройства режима ядра WIN32K.SYS, который управляет отображением окон, выводом на экран, вводом данных с клавиатуры или при помощи мыши и передачей сообщений. Он включает также библиотеку интерфейсов графических устройств (Graphical Device Interface library – GDL.DLL), используемую для создания графических объектов и текста.

Вызов Win32 API-функций. Когда приложение вызывает API-функцию из подсистемы Win32, может произойти одно из нескольких событий:

- если DLL подсистемы (например, USER32.DLL), экспортирующая данную API-функцию, содержит весь код, необходимый для выполнения функции, то функция выполняется и возвращает результат;
- API-функции, вызываемой приложением, может потребоваться вызвать для выполнения вспомогательных действий дополнительный модуль, принадлежащий подсистеме Win32 (но не той DLL, которая экспортирует данную функцию);
- API-функции, вызываемой приложением, могут понадобиться услуги недокументированного системного сервиса. Например, чтобы создать новый процесс, API-функция CreateProcess вызывает недокументированный системный сервис NTCreateProcess для реального создания данного процесса. Это делается с помощью функций библиотеки NTDLL.DLL, которая помогает осуществлять переход из пользовательского режима в режим ядра.

Исполнительная система Windows. Сервисы исполнительной системы Windows составляют низкоуровневую часть Windows NT режима ядра, включенную в файл NTOSKRNL.EXE, и делятся на две группы: исполнительную систему (executive), относящуюся к верхнему уровню, и ядро (kernel). Ядро – это самый нижний уровень операционной системы, реализующий наиболее фундаментальные сервисы, такие как: планирование потоков, обработку исключений, обработку прерываний, синхронизацию процессоров в многопроцессорной системе, создание объектов ядра.

Уровень абстрагирования от аппаратуры **(HAL)** – это библиотека режима ядра (HAL.DLL), которая реализует низкоуровневый интерфейс с аппаратурой. Компоненты Windows и драйверы устройств от других компаний взаимодействуют с

аппаратурой посредством HAL. Существует много версий HAL под различные аппаратные платформы. Подходящий уровень выбирается в процессе установки Windows.

Объекты и их дескрипторы

Архитектура Windows базируется на использовании множества различных объектов. Объект ядра (kernel object) – это структура данных, доступ к членам которой имеет только ядро Windows. Примеры объектов ядра:

- объект Process представляет процесс;
- объект Thread определяет поток;
- объект File представляет открытый файл;
- объект File-mapping представляет отображаемый в память файл (memory-mapped file), т.е. файл, содержимое которого отображено непосредственно на виртуальное адресное пространство и используется как физическая память;
- объект Pipe используется для обмена данными между процессами;
- объект Event является объектом синхронизации потоков, сигнализирующим о завершении операции;
- объект Mutex представляет собой объект синхронизации потоков, который может использоваться несколькими процессами;
- объект Semaphore используется для того, чтобы учитывать ресурсы и сигнализировать потоку о доступности ресурса на данный момент.

Кроме объектов ядра, существуют также пользовательские объекты и объекты GDI, такие как: меню, окна, шрифты, кисти и курсоры мыши.

Дескрипторы. Одной из характеристик любого объекта является дескриптор, который используется для идентификации этого объекта.

Хотя к объектам ядра нельзя получить непосредственный доступ из пользовательского режима, в Windows API есть функции, которые можно вызывать из данного режима для управления этими объектами. Это своего рода инкапсуляция (encapsulation), защищающая объекты от непредусмотренных или неразрешенных действий. Когда создается объект ядра посредством вызова соответствующей API функции (CreateProcess, Greate- Thread, CreateFile и GreateFileMapping), функция возвращает дескриптор вновь созданного объекта. Такой дескриптор может быть передан другой API-функции для того, чтобы она могла управлять данным объектом.

В общем, дескриптор объекта является зависимым от процесса (process-specific). Это означает, что он действует только в пределах данного процесса. Некоторые идентификаторы, такие как ID процесса, наоборот, являются идентификаторами системного уровня. Другими словами, область их действия – все процессы системы.

Подсчет используемости. Объект ядра принадлежит ядру Windows, а не процессу, создавшему этот объект (или любому другому процессу). Объекты могут использоваться совместно многими процессами и применяться разными способами. У каждого процесса, который работает с объектом, есть свой собственный, действующий в пределах данного процесса, дескриптор этого объекта.

С учетом этого ядро должно поддерживать подсчет используемости (usage count) каждого объекта. Ядро уничтожает объект тогда, когда его используемость становится равной нулю, но не раньше. Таким образом, процесс, создавший данный объект, может закрыть (close) его дескриптор (посредством вызова API-функции CloseHandle), но объект не будет уничтожен, если какой-то другой процесс продолжает его использовать (имеет его дескриптор). У объектов ядра есть атрибуты защиты, которые можно использовать для ограничения доступа к данным объектам. Фактически это одно из основных свойств, отличающих объекты ядра от пользовательских объектов и объектов GDI.

Совместное использование объектов несколькими процессами. Существует несколько способов совместного использования объекта несколькими процессами.

1. Наследование. Когда процесс (а точнее поток этого процесса) создает объект ядра, он может указать, что дескриптор этого объекта наследуется (inheritable) порожденными (child) процессами, которые данный родительский процесс создает впоследствии. В этой случае дескрипторы родительского и порожденного процессов одинаковы.
2. Дублирование дескриптора. Функция DuplicateHandle определяется следующим образом:

```
BOOL DuplicateHandle(
HANDLE hSourceProcessHandle,    // Дескриптор процесса-источника.
HANDLE hSourceHandle,           // Копируемый дескриптор.
HANDLE hTargetProcessHandle,    // Дескриптор процесса-приемника.
LPHANDLE IpTargetHandle,        // Указатель на дескриптор-копию.
DWORD dwDesiredAccess,          // Доступ для дескриптора-копии.
BOOL blnheriCHandle,            // Флаг наследуемости дескриптора.
DWORD dwOptions                 // Необязательные опции.
);
```

Эта функция позволяет скопировать дескриптор объекта одного процесса в другой процесс. Новый дескриптор-копия, действующий в пределах своего процесса, может иметь значение, отличное от значения дескриптора-источника, но это не существенно, так как все дескрипторы действуют только в пределах своих процессов.

3. Именованные объекты. Многим объектам ядра при их создании может быть присвоено имя. Областью действия имени является вся система. Это означает, что любой другой процесс может получить доступ к объекту по его имени (если считать, конечно, что другому процессу это имя известно). Например, последний параметр функции

```
HANDLE CreateFileMapping (      // Дескриптор отображаемого
HANDLE hFile,                   // файла.
LPSECUKITY_ATTRIBUTES           // Необязательные атрибуты
IpFileMappingAttributes,        // защиты.
```

DWORD flProtect, // Защита отображаемого
// объекта.
DWORD dwMaxiniumSizeHigh, // Старшие 32 бита размера
// объекта.
DWORD dwMaximumSizeLow, // Младшие 32 бита размера
// объекта.
LPCTSTR lpHame // Имя объекта отображения
// файла.
);

может использоваться для задания имени отображения файла.

Предположим, создан объект File-mapping с именем MyFMO. Другой процесс может вызнать функцию OpenFileMapping с этим именем в качестве последнего аргумента. Функция вернет зависимый от процесса дескриптор объекта для использования во втором процессе. В виде альтернативы второй процесс может вызвать функцию GreateFileMapping, используя имя объекта в качестве ее последнего аргумента. Система определит, что объект File-mapping с таким именем уже существует и просто вернет его дескриптор. Здесь могут возникнуть проблемы, поскольку процесс считает, что он создает новый объект, тогда как в действительности он получает дескриптор существующего объекта. Программист должен сразу проверить возвращенное функцией CreateFileMapping значение, чтобы правильно сориентироваться в ситуации.

API-функции, необходимые для отображения файла

При выполнении данной лабораторной работы необходимо изучить и использовать следующие API-функции:

1. Создание объекта FILE на базе существующего файла:

HANDLE CreateFile(
LPCTSTR lpFileName,
DWORD dwDesiredAccess,
DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
DWORD dwCreationDistribution,
DWORD dwFlagsAndAttributes,
HANDLE hTemplateFile
);

Рассмотрим параметры этой API- функции:

- lpFileName – имя открываемого файла;
- dwDesiredAccess – GENERIC_READ, чтобы разрешить чтение с устройства. GENERIC_WRITE, чтобы разрешить запись на устройство (константы можно объединить). Ноль, чтобы разрешить только получение информации об устройстве;
- dwShareMode – 0 для запрета общего доступа, FILE_SHARE_READ и/или FILE_SHARE_WRITE для разрешения общего доступа к файлу;
- lpSecurityAttributes – указатель на структуру, определяющую атрибуты безопасности файла (если они поддерживаются операционной системой);
- dwCreationDistribution – одна из следующих констант:

CREATE_NEW: создать файл. Если файл существует, происходит ошибка;
CREATE_ALWAYS: создать файл. Предыдущий файл перезаписывается;
OPEN_EXISTING: открываемый файл должен существовать (обязательно используется для устройств);
OPEN_ALWAYS: создать файл, если он не существует TRUNCATE_ EXISTING: существующий файл усекается до нулевой длины;

- dwFlagsAndAttributes Long – комбинация следующих констант:

FILE_ATTRIBUTE_ARCHIVE: установить архивный атрибут;
FILE__ATTRIBUTE_COMPRESSED: помечает файл как подлежащий сжатию или задает сжатие для файлов каталога по умолчанию;
FILE_ATTRIBUTE_NORMAL: другие атрибуты файла не заданы;
FILE_ATTRJBUTE_HIDDEN: файл или каталог является скрытым;
FILE_ATTRIBUTE_READONLY: файл доступен только для чтения;
FILE_ATTRIBUTE_SYSTEM: файл является системным;
FILE_FLAG_WRITE_THROUGH: операционная система не откладывает операции записи в файл;
FILE_FLAG_OVERLAPPED: разрешить перекрывающиеся операции с файлом;
FILE_FLAG_NO_BUFFERING: запретить промежуточную буферизацию файла. Адреса буферов должны выравниваться по границам секторов для текущего тома; FILE_FLAG_RANDOM_ACCESS: буферизация файла оптимизируется для произвольного доступа;
FILE_FLAG_SEQUENTIAL_SCAN: буферизация файла оптимизируется для последовательного доступа;
FILE_FLAG_DELETE_ON_CL0SE: при закрытии последнего открытого дескриптора файл удаляется. Идеально подходит для временных файлов;

- hTemplateFile – если параметр не равен нулю, он содержит дескриптор файла, с которого будут скопированы расширенные атрибуты нового файла. Возвращаемое значение – дескриптор файла в случае успеха. INVALID_HANDLE_VALUE при ошибке. Устанавливает информацию GetLastError. Даже если функция завершилась успешно, но файл существовал и был задан флаг CREATE_ALWAYS или OPEN_ALWAYS, GetLast-Error возвращает ERROR_ALREADY_EXISTS.

2. Создание объекта файлового отображения File-mapping:

HANDLE CreateFileMapping(
HANDLE hFile,
LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
DWORD flProtect,
DWORD dwMaximumSizeHigh,
DWORD dwMaximumSizeLow,
LPCTSTR lpName
);

Рассмотрим параметры этой API- функции:

- hFile – дескриптор файла, для которого создается отображение;
- lpFileMappingAttributes – SECUR1TY_ATTRIBLITES – объект безопасности, используемый при создании файлового отображения. NULL для использования стандартных атрибутов безопасности;
- flProtect – одна из следующих констант:

PAGE_READONLY: созданное файловое отображение доступно только для чтения;

PAGE_READWRITE: файловое отображение доступно для чтения и записи;

PAGE_WRITECQPY: разрешается копирование при записи; также в комбинацию могут включаться следующие константы: SEC_COMMГГ: для страниц секции выделяется физическое место в памяти или файле подкачки;

SEC_IMAGE: файл является исполняемым;

SEC_RESERVE: для страниц секции резервируется виртуальная память без фактического выделения.

- dwMaximumSizeHigh – максимальный размер файлового отображения (старшие 32 бита);
- dwMaximumSizeLow – максимальный размер файлового отображения (младшие 32 бита); Если параметры dwMaximumSizeHigh и dwMaximumSizeLow одновременно равны нулю, используется фактический размер файла на диске;
- lpName – имя объекта файлового отображения. Если файловое отображение с заданным именем уже существует, функция открывает его.

Возвращаемое значение – дескриптор созданного объекта файлового отображения. Ноль в случае ошибки. Устанавливает информацию GetLast-Error. Даже если функция завершилась успешно, но возвращенный манипулятор принадлежит существующему объекту фазового отображения, GetLastError возвращает ERROR_ALREADY_EXISTS. В этом случае размер файлового отображении определяется размером существующего объекта, а не параметрами функции.

3. Функция отображает объект файлового отображения в адресное пространство текущего процесса:

LPVOID MapViewOfFile(
HANDLE hFileMappingObject,
DWORD dwDesiredAccess,
DWORD dwFileOffsetHigh,
DWORD dwFileOffsetLow,
DWORD dwNumberOfBytesToMap
);

Рассмотрим параметры этой API- функции:

- hFileMappingObject – дескриптор объекта файлового отображения;
- dwDesiredAccess – одна из следующих констант:

FILE_MAP_WRITE: отображение доступно для чтения и записи. Объект файлового отображения должен быть создан с флагом PAGE_READWRITE;

FILE_MAP_READ: отображение доступно только для чтения. Объект файлового отображения должен быть создан с флагом PAGE_READ или PAGE_READWRITE; FILE_MAP_ALL_ACCESS: то же, что FILE_MAP_WRITE HLE_MAP_COPY: копирование при записи. В Windows 95 Обьект файлового отображения должен быть создан с флагом PAGE_WRITECOPY;

- dwFileOffsetHigh – старшие 32 бита смещения в файле, с которого начинается отображение;
- dwFileOffsetLow – младшие 32 бита смещения в файле, с которого начинается отображение;
- dwNumberOfBytesToMap – количество отображаемых байт в файле. Ноль, чтобы использовать весь объект файлового отображения.

Возвращаемое значение – начальный адрес отображения в памяти. Ноль при ошибке. Устанавливает информацию GetLastError.

Комментарии: dwOffsetLow и dwOffsetHigh должны содержать смещение с учетом гранулярности выделения памяти в системе. Другими словами, если гранулярность памяти в системе равна 64 Кбайт (выделяемые блоки выравниваются по границе 64 Кбайт), значение должно быть кратно 64 Кбайт. Чтобы получить гранулярность выделения памяти в системе, воспользуйтесь функцией GetSystemlnfo. В большинстве приложений передается ноль, чтобы отображение начиналось с начала файла. Параметр IpBaseAddress также должен быть кратен значению гранулярности.

4. Функция закрывает объект ядра. К числу объектов ядра относятся объекты файлов, файловых отображений, процессов, потоков, безопасности и синхронизации:

BOOL CloseHandle(
HANDLE hObject // Дескриптор закрываемого объекта.
);

Возвращаемое значение – ненулевое значение в случае успеха, ноль при ошибке. Устанавливает информацию GetLastError.

Объекты ядра удаляются лишь после того, как будут закрыты все ссылки на них.

5. Функция прекращает отображение объекта в адресное пространство текущего процесса:

```
BOOL UnmapViewOfFile(
LPCVOID lpBaseAddress        // Базовый адрес отображения,
                             // установленного ранее функцией
                             // MapViewOfFile.
);
```

ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. Провести исследование ОС с использованием системного монитора:
   1) Определить количество процессов, потоков, дескрипторов в ОС, изменить их число, запуская на выполнение новые приложения.
   2) Определить процент работы в пользовательском режиме (% User Time), процент работы в привилегированном режиме (% Privileged Time) и процент времени бездействия при выполнении, связанными с интенсивными графическими операциями.
   3) Включить в отчет полученные графики и привести их объяснение.
2. Разработать программу, реализующую следующую задачу:
   1) Создать текстовый файл (можно с использованием notepad).
   2) Создасть объект File на базе созданного в предыдущем пункте файла, используя API-функцию CreateFile. Вывести значение дескриптора объекта File.
   3) Используя дескриптор объекта File-mapping, а также API-функцию MapViewOf File, отобразить части файла в память. Данная функция назначает область виртуальной памяти, выделяемой этому файлу. Базовый адрес выделенной области памяти является дескриптором представления этой области в виде отображения файла.
   4) Используя базовый адрес и функцию CopyMemory, прочитайте информацию из отображаемого файла. Измените регистр текста в тестовом файле, и запишите информацию в этот же файл.
   5) Закрыть все дескрипторы.

На экране должны фиксироваться все этапы работы созданного приложения.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое режим ядра и пользовательский режим?
2. Перечислите объекты ядра ОС.
3. Понятие сервиса ОС. Какие системные процессы вы знаете?
4. Опишите работу ядра ОС в привилегированном режиме.
5. Какие API-функции необходимы для отображения файла в адресное пространство?
6. Какая API-функция возвращает начальный адрес области отражения файла?

Лабораторная работа 3

АРХИТЕКТУРА ПАМЯТИ WINDOWS

Цель работы: получение практических навыков по использованию Win32 API для исследования памяти Windows.

ОСНОВНЫЕ ПОЛОЖЕНИЯ

Типы памяти

Физическая память – это реальные микросхемы RAM, установленные в компьютере. Каждый байт физической памяти имеет физический адрес, который представляет собой число от нуля до числа на единицу меньшего, чем количество байтов физической памяти. Например, ПК с установленными 64 Мб RAM имеет физические адреса &H00000000-&H04000000 в шестнадцатеричной системе счисления, что в десятичной системе будет 0-67 108 863. Физическая память (в отличие от файла подкачки и виртуальной памяти) является исполняемой, т.е. памятью, из которой можно читать и в которую центральный процессор может посредством системы команд записывать данные.

Виртуальная память – это просто набор чисел, о которых говорят как о виртуальных адресах. Программист может использовать виртуальные адреса, но Windows не способна по этим адресам непосредственно обращаться к данным, поскольку такой адрес не является адресом реального физического запоминающего устройства, как в случае физических адресов и адресов файла подкачки. Для того чтобы код с виртуальными адресами можно было выполнить, такие адреса должны быть отображены на физические адреса, по которым действительно могут храниться коды и данные. Эту операцию выполняет диспетчер виртуальной памяти (Virtual Memory Manager – VMM). Операционная система Windows обозначает некоторые области виртуальной памяти как области, к которым можно обратиться из программ пользовательского режима. Все остальные области указываются как зарезервированные. Какие области памяти доступны, а какие зарезервированы, зависят от версии операционной системы (Windows 9x или Windows NT).

Взаимосвязь виртуального адресного пространства процесса с физической и внешней памятью представлена на рис. 2.

Страничные блоки памяти. Как известно, наименьший адресуемый блок памяти – байт. Однако самым маленьким блоком памяти, которым оперирует Windows VMM, является страница памяти, называемая также страничным блоком памяти. На компьютерах с процессорами Intel объем страничного блока равен 4 Кб.

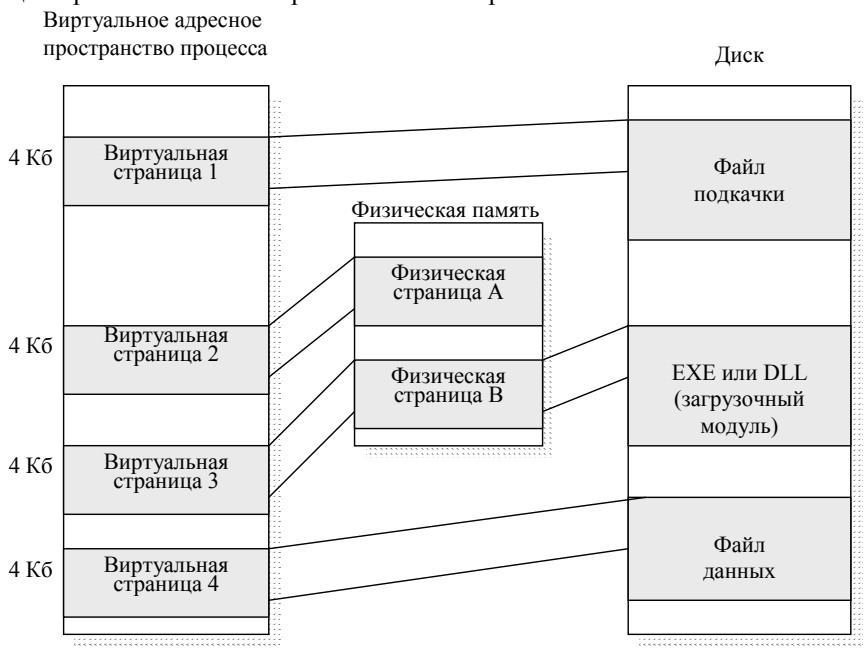

Рис. 2. Взаимосвязь виртуального адресного пространства процесса с физической и внешней памятью

Память файла подкачки. Страничный файл, который называется также файлом подкачки, в Windows находится на жестком диске. Он используется для хранения данных и программ точно так же, как и физическая память, но его объем обычно превышает объем физической памяти. Windows использует файл подкачки (или файлы, их может быть несколько) для хранения информации, которая не помещается в RAM, производя, если нужно, обмен страниц между файлом подкачки и RAM.

Таким образом, диапазон виртуальных адресов скорее согласуется с адресами в файле подкачки, чем с адресами физической памяти. Когда такое согласование достигается, говорят, что виртуальные адреса спроецированы на файл подкачки, или являются проецируемыми на файл подкачки.

Набор виртуальных адресов может проецироваться на физическую память, файл подкачки или любой файл.

Файлы, отображаемые в память. В лабораторной работе 2 обсуждались файлы, отображаемые в память, там же был приведен пример отображения файла. Любой файл применяется для проецирования виртуальной памяти так же, как для этих целей используется файл подкачки. Фактически единственное назначение файла подкачки – проецирование виртуальной памяти. Поэтому файлы, проецируемые в память подобным образом, называются отображаемыми в память. На рисунке 2 изображены именно такие файлы. Соответствующие виртуальные страницы являются спроецированными на файл.

Функция CreateFileMapping создает объект "отображение файла" используя дескриптор открытого файла, и возвращает дескриптор этого объекта. Дескриптор может использоваться с функцией MapViewOfFile, отображающей файл в виртуальную память.

Начальный адрес объекта "отображение файла" в виртуальной памяти возвращает функция MapViewOfFile. Можно также сказать, что представление проецируется на файл с дескриптором hFile.

Если параметр hFile, передаваемый функции CreateFileMapping, установлен в -1, то объект "отображение файла" (любые представления, созданные на основе этого объекта) проецируем на файл подкачки, а не на заданный файл.

Совместно используемая физическая память. О физической памяти говорят, что она совместно используется, если она отображается на виртуальное адресное пространство нескольких процессов, хотя виртуальные адреса в каждом процессе могут отличаться.

Если файл, такой как DLL, находится в совместно используемой физической памяти, то о нем можно говорить как о совместно используемом.

Одно из преимуществ файлов, отображаемых в память, заключается в том, что их легко использовать совместно. Присвоение имени объекту "отображение файла" делает возможным совместное использование файла несколькими процессами. В этом случае его содержимое отображено на совместно используемую физическую память. Возможно также совместное пользование содержимого файла подкачки с помощью механизма отображения файла.

Адресное пространство процесса. Каждый процесс Win32 получает виртуальное адресное пространство (virtual address space), называемое также адресным пространством, или пространством процесса, объем которого равен 4 Гб. Таким образом, код процесса может ссылаться на адреса с &H00000000 по &HFFFFFFFF (или с 0 по $2^{32} – 1 = 4\ 294\ 967\ 295$ в десятичной системе счисления). Конечно, так как виртуальные адреса – это просто числа, заявление о том, что каждый процесс получает свое собственное виртуальное адресное пространство, выглядит довольно бессмысленным.

Тем не менее, это утверждение должно означать, что Windows не видит никакой взаимосвязи в том, что и процесс *A*, и процесс *B* используют один и тот же виртуальный адрес, например &H40000000. В частности, Windows может сопоставить (или не сопоставить) виртуальным адресам каждого процесса разные физические адреса.

Использование адресного пространства в Windows 9x

Общая схема использования адресного пространства процесса в Windows 9x показана на рис. 3.

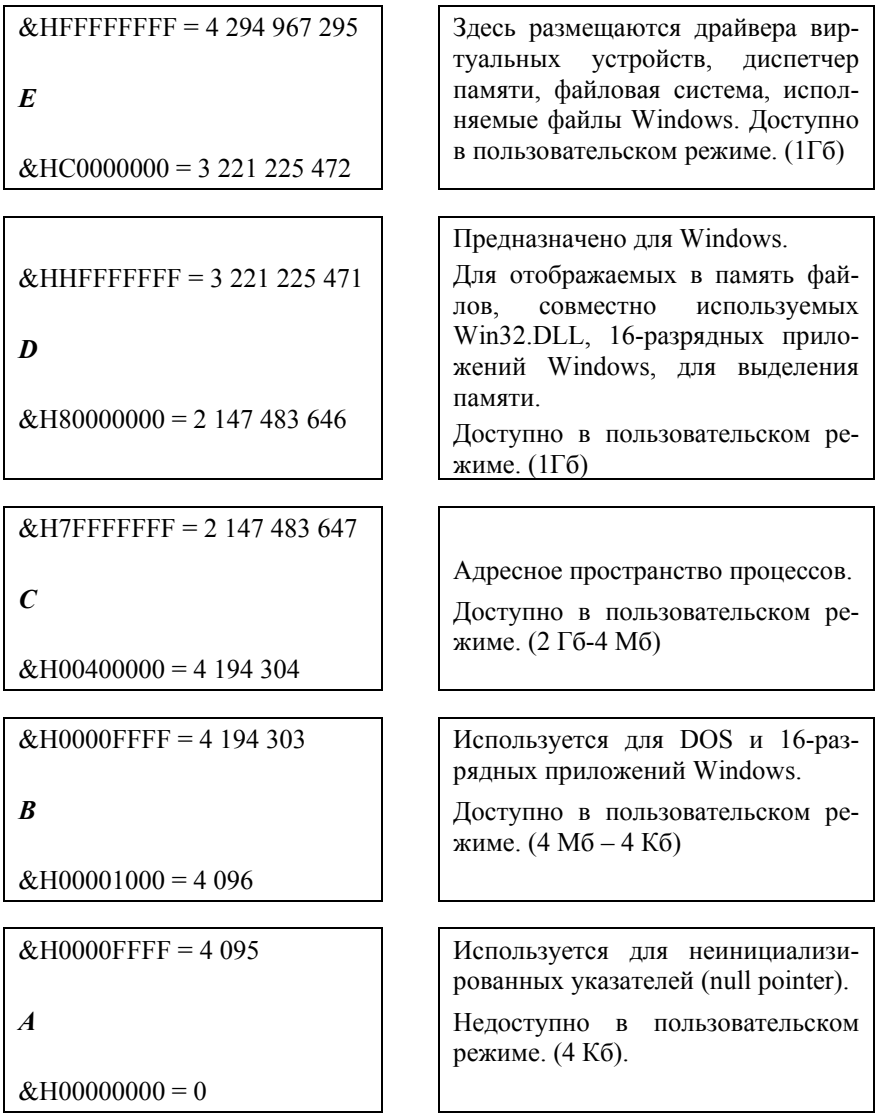

Рис. 3. Схема использования адресного пространства процесса

Область A. Как следует из рисунка, Windows 9x резервирует область *A*, объем которой всего лишь 4 Кб – с целью предупреждения о нулевых указателях. Эта область защищена, и попытка обращения к ней из программы пользовательского режима приводит к ошибке нарушения доступа.

Область B. Данная область памяти используется для поддержания совместимости с приложениями DOS и 16-разрядными приложениями Windows. Несмотря на потенциальную доступность, она не должна использоваться для программирования.

Область C. Область *C* – это адресное пространство, используемое прикладными программами и их DLL. Здесь размещаются также и модули Windows. Например, если приложению требуется управляющий элемент OCX, его модуль будет находиться в этой области.

Область D. Windows 9x отображает системные DLL Win32 (KERNEL32.DLL, USER32.DLL и т.д.) в это адресное пространство. Данные файлы используются совместно, т.е. несколько процессов могут обращаться к единственной копии такого файла в физической памяти. Область *D* доступна для программ пользовательского режима (однако размещать их здесь не рекомендуется).

Область E. Данная область также содержит совместно используемые файлы Windows, такие как исполнительная система Windows и ядро, драйверы виртуальных устройств, файловая система, программы управления памятью. Она также доступна для программ пользовательского режима.

Распределение виртуальной памяти

Каждая страница виртуального адресного пространства может находиться в одном из трех состояний:

1) Reserved (зарезервирована) – страница зарезервирована для использования;
2) Committed (передана) – для данной виртуальной страницы выделена физическая память в файле подкачки или в файле, отображаемом в память;
3) Free (свободна) – данная страница не зарезервирована и не передана, и поэтому в данный момент она недоступна для процесса.

Виртуальная память может быть зарезервирована или передана с помощью вызова API-функции VirtualAlloc:

```
LPVOID VirtualAlloc(
LPVOID IpAddress,                 // Адрес резервируемой или
                                  // выделяемой области.
DWORD dwSise,                     // Объем области.
DWORD flAllocationType,           // Тип распределения.
DWORD flProtect                   // Тип защиты от доступа.
);
```

Параметр flAllocationType может принимать значения следующих констант (помимо других):

1) MEM_RESERVE – параметр, резервирующий область виртуального адресного пространства процесса без выделения физической памяти. Тем не менее, память может быть выделена при следующем вызове этой же функции;

2) MEM_COMMIT – параметр, выделяющий физическую память в оперативной памяти или в файле подкачки на диске для заданного зарезервированного набора страниц.

Эти две константы могут объединяться для того, чтобы зарезервировать и выделить память одной операцией.

Разделение процедур резервирования и передачи памяти имеет некоторые преимущества. Например, резервирование памяти является очень полезной процедурой с точки зрения практичности. Если приложению требуется большой объем памяти, можно зарезервировать всю память, а выделить только ту часть, которая нужна в данный момент, раздвигая, таким образом, временные рамки более трудоемкой операции выделения физической памяти.

Windows тоже использует этот подход, когда выделяет память под стек каждого вновь создаваемого потока. Система резервирует 1 Мб виртуальной памяти под стек каждого потока, но выделяет первоначально только две страницы (8 Кб).

Защита памяти

Параметр flProtect функции virtualAlloc используется для задания типа защиты от доступа, соответствующего вновь выделенной виртуальной памяти (это не относится к резервируемой памяти). Существуют следующие методы защиты:

1) PAGE_READONLY присваивает доступ "только для чтения" выделенной виртуальной памяти;
2) PAGE_READWRITE назначает доступ "чтение-запись" выделенной виртуальной памяти;
3) PAGE_WRITECOPY устанавливает доступ "запись копированием" (copy-onwrite) выделенной виртуальной памяти.
4) PAGE_EXECUTE разрешает доступ "выполнение" выделенной виртуальной памяти. Тем не менее, любая попытка чтения – записи этой памяти приведет к нарушению доступа;
5) PAGE_EXECUTE_READ назначает доступ "выполнение" и "чтение";
6) PAGE_EXECUTE_READWRITE разрешает доступ "выполнение", "чтение" и "запись";
7) PAGE_EXECUTE_WRITECOPY присваивает доступ "выполнение", "чтение" и "запись копированием";
8) PAGE_NOACCESS запрещает все виды доступа к выделенной виртуальной памяти.

Любые из этих значений, за исключением PAGE_NOACCESS, могут комбинироваться при помощи логического оператора OR со следующими двумя флагами:

1) PAGE_GUARD определяет помеченные страницы как защищенные (guard page). При любой попытке обращения к защищенной странице система возбуждает исключительную ситуацию STATUS_GUARD_PAGE и снимает с данной страницы статус защищенной. Таким образом, защищенные страницы предупреждают только о первом обращении к ним;
2) PAGE_NOCACHES запрещает кэширование выделенной памяти.

Следует знать, что такое доступ "запись копированием". Допустим, некоторая страница физической памяти совместно используется двумя процессами. Если она помечена как "только для чтения", то два процесса без проблем могут совместно пользоваться этой страницей. Однако возможны ситуации, когда каждому процессу требуется разрешить запись в эту память, но без воздействия на другой процесс. После установки защиты "запись копированием" при попытке записи в совместно используемую страницу система создаст ее копию специально для процесса, которому нужно осуществить запись. Таким образом, данная страница перестает быть совместно используемой, а представление ее данных в других процессах остается неизменным.

Необходимо отметить, что атрибуты защиты страницы могут быть изменены с помощью API-функции Virtual Protect.

Пример использования функции GlobalMemoryStatus

API-функция GlobalMemoryStatus, записывающаяся таким образом:

```
procedure GlobalMemoryStatus(var lpBuffer: TMemoryStatus); stdcall;
procedure GlobalMemoryStatus; external kernel32 name 'GlobalMemory-Status';
```

выводит множество данных, имеющих отношение к памяти, в составе следующей структуры:

```
struct _MEMORYSTATUS {            // Размер структуры
DWORD dwLength;                   // MEMORYSTATUS.
DWORD dwMernoryLoad;              // Процент используемой памяти.
DWORD dwTotalPhys;                // Количество байтов физической
                                  // памяти.
DWORD dwAvailPhys;                // Количество свободных байтов
                                  // физической памяти.
DWORD dwTotalPageFile;            // Размер в байтах файла
                                  // подкачки.
DWORD dwAvailPageFile;            // Количество свободных байтов
                                  // файла подкачки.
DWORD dwTotalVirtual;             // Количество байтов адресного
                                  // пространства,
```

```
                                    // доступного пользователю.
DWORD dwAvailvirtual;               // Количество свободных байтов
                                    // памяти, доступных пользователю.
}
```

Преобразование виртуальных адресов в физические: попадание

Процесс преобразования при отображении виртуальных адресов в физические, называемый попаданием в физическую страницу, показан на рис. 4.

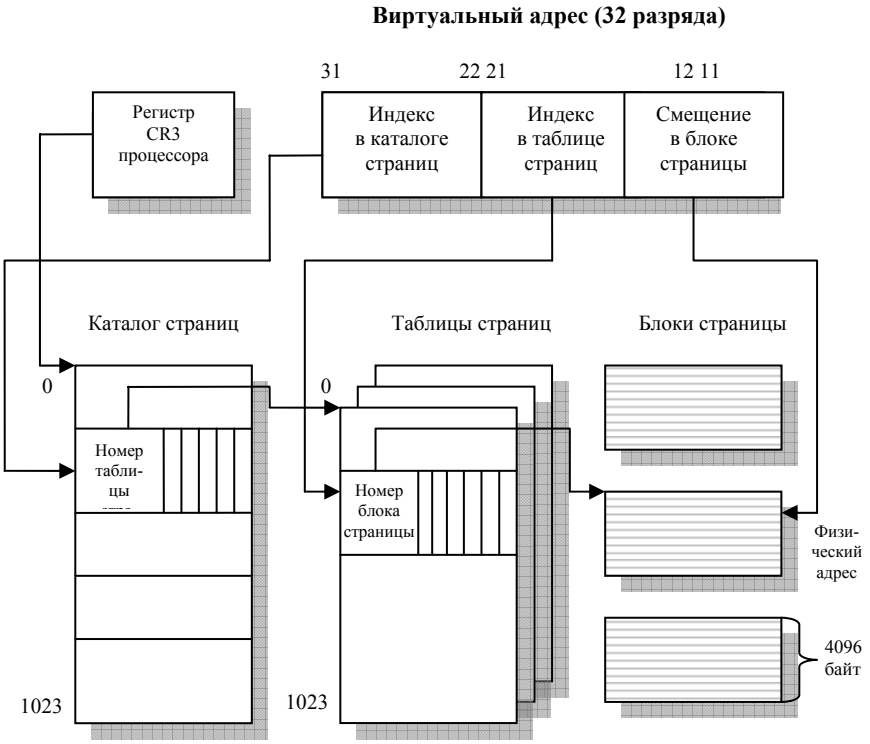

Рис. 4. Связь виртуальной памяти с физической

Все виртуальные адреса делятся на три части. Самая левая часть (биты 22-31) содержит индекс каталога страниц процесса. Windows поддерживает отдельный каталог страниц для каждого процесса. Его адрес хранится в одном из регистров центрального процессора, который называется CR3. (В операцию переключения задач входит переведение CR3 в состояние, когда он указывает на каталог страниц процесса, на который осуществляя переключение.) Каталог страниц одержит 1024 четырехбайтовых элемента.

Windows поддерживает для каждого процесса совокупность таблиц страниц. Каждый элемент каталога страниц содержит уникальный номер. Поэтому Windows поддерживает до 1024 таблиц страниц. (В действительности таблицы страниц создаются только при попытке обращения к данным или коду конкретному виртуальному адресу, а не когда выделяется виртуальная память).

Следующая часть виртуального адреса (биты 12-21) используется в качестве индекса в таблице страниц, соответствующей выбранному элементу каталога страниц. Каждый элемент таблицы, соответствующий указанному индексу, содержит в 20 старших разрядах номер страничного блока, который задает конкретный страничный блок физической памяти.

Третья, и последняя, часть виртуального адреса (биты 0-11) представляет собой смещение в данном страничном блоке. Сочетание номера страничного блока и смещения дают в совокупности адрес физической памяти.

Так как каждая таблица страниц состоит из 1024 элементов и количество таблиц равно 1024, общее количество страничных блоков, которое можно определить таким образом, будет $1024 \times 1024 = 2^{10} \times 2^{10} = 2^{20}$. Так как каждый страничный блок имеет объем 4 Кб = 4×2^{10} байт, то теоретический предел физического адресного пространства будет $4 \times 2^{30} = 4$ Гб.

У этой довольно сложной схемы преобразования есть несколько важных преимуществ. Одно из них – очень небольшой объем страничных блоков, которые легко могут быть размещены в памяти. Гораздо легче найти непрерывный блок памяти размером 4 Кб, чем, скажем, 64 Кб.

Но основное преимущество заключается в том, что адреса виртуальной памяти двух процессов могут быть сознательно преобразованы в разные или в одни и те же физические адреса.

Предположим, что Process1 и Process2 обращаются в программе к одному и тому же виртуальному адресу. При преобразовании виртуальных адресов в физические для каждого из процессов используются их собственные каталоги страниц. Поэтому, хотя индексы в каталогах страниц одинаковы и в том, и в другом случаях, они все же представляют собой индексы из разных каталогов. Таким способом VMM может гарантировать, что виртуальные адреса каждого процесса будут преобразованы в разные физические адреса.

С другой стороны, VMM может также дать гарантию, что виртуальные адреса двух процессов, независимо от того являются ли они одинаковыми или нет, будут преобразованы в один и тот же физический адрес. Один из способов добиться

этого – установить соответствующий элемент в обоих каталогах страниц на одну и ту же таблицу страниц и, следовательно, на один и тот же страничный блок. Таким образом, процессы могут совместно использовать физическую память.

Кучи памяти в 32-разрядной ОС Windows

При создании процесса Windows назначает ему кучу по умолчанию, т.е. изначально резервирует область виртуальной памяти объемом 1 Мб. Тем не менее, при необходимости система будет регулировать размер кучи, которая используется самой Windows для различных целей.

API-функция GetProcessHeap используется для получения дескриптора кучи. При помощи функции HeapCreate, возвращающей дескриптор кучи, программист может создавать дополнительные кучи.

Есть несколько причин создавать дополнительные кучи вместо того, чтобы использовать кучу по умолчанию. Например, те кучи, которые предназначены для конкретных задач, часто оказываются более эффективными. Кроме того, ошибки записи данных в кучу, память для которой выделена из специализированной кучи, не затронут данных других куч. Наконец, выделение памяти из специализированной кучи в общем случае будет означать, что данные в памяти упакованы более плотно друг к другу, а это может уменьшить потребность в загрузке страниц из файла подкачки. Следует также упомянуть, что доступ к куче упорядочен, т.е. система заставляет каждый поток, пытающийся обратиться к памяти кучи, дожидаться своей очереди, пока другие потоки не закончат производимые операции. Следовательно, только один поток в каждый момент времени может выделять или освобождать память кучи во избежание неприятных конфликтов.

Функции работы с кучей

Для работы с кучами используются следующие функции:

1. GetProcessHeap возвращает дескриптор кучи процесса по умолчанию;
2. GetProcessHeaps возвращает список дескрипторов всех куч, используемых в данный момент процессом.
3. HeapAlloc выделяет блок памяти из заданной кучи.
4. HeapCompact дефрагментирует кучу, объединяя свободные блоки; может также освобождать неиспользуемые страницы памяти кучи.
5. HeapCreate создает новую кучу в адресном пространстве процесса.
6. HeapDestroy удаляет заданную кучу.
7. HeapFree освобождает предварительно выделенные блоки памяти кучи.
8. HeapLock блокирует кучу, при использовании данной функции только один поток имеет к ней доступ. Другие потоки, запрашивающие доступ, переводятся в состояние ожидания до тех пор, пока поток, владеющий кучей, не разблокирует ее. Это одна из форм синхронизации потоков, т.е. тот прием, которым система реализует упорядоченность доступа.
9. HeapReAlloc перераспределяет блоки памяти кучи. Используется для изменения размера блока.
10. Heapsize возвращает размер выделенного блока памяти кучи.
11. HeapUnlock разблокирует кучу, которая до этого была заблокирована функцией HeapLock.
12. HeapValidate проверяет пригодность кучи (или отдельного ее блока), если имеются ли какие-либо повреждения.
13. HeapWalk позволяет программисту просматривать содержимое кучи. Обычно используется при отладке.

Отображения виртуальной памяти

Функция Win32 API VirtualQuery может использоваться для получения информации о состоянии адресов виртуальной памяти. Синтаксис ее таков:

```
DWORD VirtualQuery(
LPCVOID IpAddress,                      // Адрес области.
PMEMORY_BASICONFORMATION IpBuffer,      // Адрес информационного
                                        // буфера.
DWORD dwLength                          // Размер буфера.
);
```

Используется также функция VirtualQueryEx, расширенная версия VirtualQuery, которая позволяет получать информацию о внешних виртуальных адресных пространствах:

```
DWORD VirtualQueryEx(
HANDLE hProcess                         // Дескриптор процесса.
LPCVOID IpAddress,                      // Адрес области.
MEMORY_BASIC_INFORMATION IpBuffer,      // Адрес информационного
                                        // буфера.
DWORD dwLength                          // Размер буфера.
);
```

Параметр hProcess – это дескриптор процесса. Параметр IpAddress – это начальный адрес для записи результирующих данных, который будет округляться в меньшую сторону до ближайшего кратного размеру страницы (4 Кб). Обе функции возвращают информацию в следующую структуру:

```
Struct MEMORY_BASIC_INFORMATION {
PVOID BaseAdciress;                     // Базовый адрес области.
PVOID AllocationBase;                   // Базовый адрес выделенной
                                        // области.
DWORD AllocationProtect;                // Первоначальная защита от
                                        // доступа.
```

DWORD RegionSize; // Размер области в байтах.
DWORD State; // Передана зарезервирована,
// свободна.
DWORD Protect; // Текущая защита от доступа.
DWORD Type; // Тип страниц.
}

Функция VirtualQueryEx всегда заполняет следующие члены структуры MEMORY_BASIC_INFORMATION:

- BaseAddress, которая возвращает базовый адрес заданной страницы;
- RegionSize, представляющая собой количество байтов от начала заданной страницы до вершины заданной области.

Если страница, содержащая адрес IpAddress, свободна (не зарезервирована и не передана), член структуры State содержит символьную константу MEM_FREE. Остальные члены (кроме BaseAddress и RegionSize*)* не имеют значения.

Если страница, содержащая адрес IpAddress, не свободна, функция определяет выделенную область (allocation region), т.е. область виртуальной памяти, которая включает заданную страницу и была, первоначально выделена с помощью вызова функции VirtualAlloc.

ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Разработать программу, которая:

1) выдает информацию, получаемую при использовании API GlobalMemoryStatus (при выводе информации использовать диаграммы);
2) составляет карту виртуальной памяти для любого процесса.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Перечислите типы адресов в ОС Windows.
2. Как от адресов виртуальной памяти перейти к физической?
3. Понятие кучи памяти в ОС Windows.
4. Что такое кэш-память?
5. Что такое адресное пространство процесса?
6. Перечислите области, которые присутствуют в адресном пространстве ОС Windows.
7. Какие алгоритмы распределения памяти вы знаете?

Лабораторная работа 4

ПРОЦЕССЫ

Цель работы: получение практических навыков при использовании Win32 API для исследования процессов.

ОСНОВНЫЕ ПОЛОЖЕНИЯ

Процесс – это исполняемый экземпляр приложения и набор ресурсов, которые выделяются данному исполняемому приложению и включают в себя следующее:

1) виртуальное адресное пространство;
2) системные ресурсы, такие как растровые изображения, файлы, области памяти и т.д.;
3) модули процесса, т.е. исполняемые модули, которые отображены (загружены) в его адресное пространство. Это могут быть динамические библиотеки (DLI), драйверы (DRV) и управляющие элементы (OCX), основной загрузочный модуль (EXE) процесса, который иногда и называют собственно модулем. Модуль данных (или программный модуль) может или находиться на диске, или быть загруженным в физическую память (RAM). Правда, термин "загружен" (loaded) имеет иное значение, относящееся к виртуальному адресному пространству процесса. Здесь больше подходит термин "отображен" (mapped), так как само отображение – это просто назначение виртуальным адресам физических адресов. После того как модуль загружен в физическую память, его физические адреса могут отображаться в различные виртуальные адресные пространств, при этом возможно использование в каждом процессе разных виртуальных адресов. Отображение не обязательно требует физического перемещения реальных данных или программ;
4) уникальный идентификационный номер, называемый идентификатором процесса;
5) один или несколько потоков управления.

Поток – это внутренняя составляющая процесса, которой операционная система выделяет процессорное время. Каждый процесс должен иметь, по крайней мере, один поток. Поток включает:

1) текущее состояние регистров процессора;
2) два стека, один из которых используется при выполнении в режиме ядра, второй – при выполнении в пользовательском режиме;
3) участок памяти для работы подсистем, библиотеки времени выполнения;
4) динамические библиотеки;
5) уникальный идентификатор, называемый идентификатором потока.

Состояние регистров, содержимое стека и области памяти называют контекстом потока (thread's context).

Основное назначение потоков – дать процессу возможность поддерживать несколько ветвей управления, т.е. выполнять больше задач одновременно. В многопроцессорной конфигурации (компьютер с двумя и более процессорами) Windows NT (но не Windows 9x) может назначать разные потоки разным процессорам в различные моменты времени, обеспечивая дейст-

вительно параллельную обработку. В однопроцессорной конфигурации процессор должен выделять кванты времени каждому исполняемому в данный момент потоку.

Дескрипторы и идентификаторы процессов

Разницу между дескрипторами и идентификаторами процессов можно установить, анализируя API-функцию CreateProcess:

BOOL CreateProcess
LPCTSTR lpApplicationName
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation;

Рассмотрим параметры API функцию CreateProcess:

- lpApplicationName, lpCommandLine – определяют имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу;
- lpProcessAttributes, lpThreadAttributes – эти параметры позволяют определить нужные атрибуты защиты для объектов "процесс" и "поток" соответственно. В эти параметры можно занести NULL, и система закрепит за данными объектами дескрипторы защиты по умолчанию. В качестве альтернативы можно объявить и инициализировать две структуры SECURITY_ ATTRIBUTES; тем самым создать и присвоить объектам "процесс" и "поток" свои атрибуты защиты. Вид структуры SECURITY ATTRIBUTES:

typedef struct _SECURITY_ATTRIBUTES {
DWORD nLength;
LPVOID lpSecurityDescriptor;
BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;

- bInheritHandles – указывает, наследовал ли новый процесс дескрипторы от процесса запроса. Если ИСТИНА, каждый наследственный открытый дескриптор в процессе запроса унаследован новым процессом;
- dwCreationFlags – определяет флаги, влияющие на то, как именно создается новый процесс. Имеется десять флагов, которые могут комбинироваться булевым оператором OR; среди них:
  - lpEnvironment – указывает на блок памяти, хранящий строки переменных окружения, которыми будет пользоваться новый процесс. Обычно вместо этого параметра передается NULL, в результате чего дочерний процесс наследует строки переменных окружения от родительского процесса;
  - lpCurrentDirectory – позволяет установить текущие диски каталог. Если его значение – NULL, рабочий каталог нового процесса будет тем же, что и у приложения, его породившего;
  - lpStartupInfo – этот параметр указывает на структуру STARTUPINFO.

Элементы структуры STARTUPINFO используются Windows-функциями при создании процесса. При этом большинство приложений порождает процессы с атрибутами по умолчанию. Но в этом случае необходимо инициализировать все элементы структуры STARTUPINFO, хотя бы нулевыми значениями, а в элемент cb – заносить размер этой структуры.

- lpProcessInformation – указывает на структуру PROCESS_INFORMATION.

Таким образом, между дескриптором и идентификатором процесса (или потока) существуют следующие основные различия:

1. Дескриптор действует в пределах своего процесса, в то время как идентификатор работает на системном уровне, и действует только в пределах того процесса, в котором он был создан.
2. У каждого процесса только один идентификатор, но может быть несколько дескрипторов.
3. Некоторым API-функциям требуется идентификатор, в то время как другим – дескриптор процесса.

Следует подчеркнуть, что, хотя дескриптор является зависимым от процесса, один процесс может иметь дескриптор другого процесса. Иными словами, если процесс ***A*** имеет дескриптор процесса ***B***, то этот дескриптор идентифицирует процесс ***B***, но действует только в процессе ***A***. Он может использоваться в процессе ***A*** для вызова некоторых API-функций, которые имеют отношение к процессу ***B.*** (Однако память процесса ***B*** остается недоступной для процесса ***A***).

Дескрипторы модулей. Каждый модуль (DLL, OCX, DRV и т.д.), загруженный в пространство процесса, имеет свой дескриптор (module handle), называемый также логическим номером экземпляра (instance handle). В 16-разрядной Windows данными терминами обозначались разные объекты, в 32-разрядной системе это один и тот же объект.

Дескриптором исполняемого модуля является начальный или базовый адрес данного исполняемого модуля в адресном пространстве процесса. Это объясняет, почему такой дескриптор имеет смысл только в рамках процесса, включающего данный модуль.

Дескрипторы модулей часто используются для вызова API-функций, которые выделяют ресурсы данному процессу. Например, функция LoadBitmap загружает ресурс растровой картинки из загрузочного файла данного процесса. Ее декларация записывается так:

HBITMAP LoadBitmap(

```
HINSTANCE hInstance,          // Дескриптор экземпляра приложения.
LPCTSTR IpBitmapeName         // Адрес имени ресурса растровой
                              // картинки.
);
```

Идентификация процесса. В программировании API, связанном с идентификацией процессов, часто используются четыре объекта:

1) идентификатор процесса;
2) дескриптор процесса;
3) полное имя загрузочного файла;
4) дескриптор модуля загрузочного файла процесса.

Получение дескриптора процесса по его идентификатору. По идентификатору можно определить дескриптор любого процесса с помощью функции OpenProcess:

```
HANDLE OpenProcess(
DWORD dwDesiredAccess,        // Флаг доступа.
BOOL bInheritHandle,          // Флаг наследования дескриптора.
DWORD dwProcessID             // Идентификатор процесса.
);
```

- dwDesiredAccess – имеет отношение к правам доступа и может принимать различные значения:

PROCESS_ALL_ACCESS эквивалентно установке флагов полного доступа;

PROCESS_DUP_HANDLE использует дескриптор как исходного процесса, так и принимающего в функции DuplicateHandle для копирования (дублирования) дескриптора;

PROCESS_QUERY_INFORMATTON задействует дескриптор процесса для чтения информации из объекта Process;

PROCESS_VM_OPERATION использует дескриптор процесса для модификации виртуальной памяти процесса;

PROCESS_TERMINATE работает для завершения процесса с его дескриптором в функции TerminateProcess;

PROCESS_VM_READ применяет для чтения из виртуальной памяти процесса его дескриптор в функции ReadProcessMemory;

PROCESS_VM_WRITE использует для записи в виртуальную память процесса его дескриптор в функции WriteProcessMernory;

SYNCHRONIZE (Windows NT) работает с дескриптором процесса в любой из функций ожидания, таких как WaitForSingleObject, для ожидания завершения процесса.

- bInheritHandle – установлен в значение True, для того чтобы позволить порожденным процессам наследовать дескриптор. Иначе говоря, порожденный процесс получает дескриптор родительского процесса. Отметим, что значение дескриптора может изменяться.
- dwProcessID – должен иметь значение идентификатора того процесса, дескриптор которого нужно узнать. Функция OpenProcess возвращает дескриптор указанного процесса.

Полученный дескриптор обязательно должен быть закрыт, если он больше не нужен. Это делается путем вызова функции CloseHandle:

```
BOOL CloseHandle(
HANDLE hObject                // Дескриптор закрываемого объекта.
);
```

Имена файлов и дескрипторы модулей. Перейти от имени файла модуля к дескриптору модуля и наоборот не составляет особого труда, по крайней мере, в пределах одного процесса. Функция GetModuleFileName принимает дескриптор модуля, чтобы вернуть полное имя (имя и путь) исполняемого файла:

```
DWORD GetModuleFileName(
HMODULE hModule,              // Дескриптор модуля.
LPTSTR IpFilename,            // Указатель на буфер-приемник
                              // пути к модулю.
DWORD nSize                   // Размер буфера в символах.
);
```

Эта функция работает только из того процесса, дескриптор модуля которого задается. Ее нельзя использовать из другого процесса.

Функция GetModuleHandle, которая также применяется внутри одного процесса, выполняет обратное действие – возвращает дескриптор модуля по имени файла (без пути):

```
HMODULE GetModuleHandle(
LPCTSTR IpModuleName          // Имя модуля.
);
```

С использованием приведенных API-функций можно разработать функцию, которая, принимая дескриптор, имя или полное имя модуля, возвращает другие два элемента в своих выходных параметрах. Возможны четыре варианта работы этой функции, например:

1. Получить дескриптор, имя и полное имя EXE:

```
Handle: &HFFFFFFFF
Name: VB6.EXE
```

FName: G:\Visual Studio\VB98\VB6.EXE

2. Получить имя и полное имя модуля:

Задаем: Handle: &H77E70000
Получаем:
Name: User32.dll
FName: G:\WINNT\system32\USER32.DLL

3. Получить дескриптор и полное имя:

Задаем: Name: User32.dll
Получаем:
Handle: &H77E70000
FName: G:\WINNT\system32\USER32.DLL

4. Получить дескриптор и имя:

Задаем: FName: G:\WINNT\system32\USER32.DLL
Получаем:
Handle: &H77E70000
Name: User32.dll

Получение идентификатора текущего процесса. Чтобы получить идентификатор текущего процесса, можно использовать функцию GetCurrentProcessId, объявление которой выглядит так:

DWORD GetCurrentProcess ID(VOID);

Эта функция возвращает идентификатор процесса. Значение идентификатора процесса может быть в верхнем диапазоне *unsigned long*, поэтому может потребоваться преобразование возвращаемого значения. Следует также учесть, что данная функция работает только в текущем процессе. Не существует способа определить идентификатор другого процесса, кроме как получить список всех процессов и выбрать из него тот процесс, характеристики которого требуются.

Получение идентификатора процесса от окна. Существует функция FindWindow. Она объявляется следующим образом:

```
HWND FindWindow(
LPCTSTR IpClassName,          // Указатель на имя класса.
LPCTSTR IpWindowName          // Указатель на имя окна.
);
```

Функция использует имя класса или заголовок окна для получения дескриптора окна. Имея дескриптор, можно вызвать функцию GetWindowThreadProcessId, возвращающую идентификатор потока, который создал данное окно, и идентификатор процесса, которому принадлежит данный поток. Синтаксис выглядит так:

```
DWORD GetWindowThreadProcessId(
HMD hWnd,                     // Дескриптор окна.
LPDWORD IpdwProcessId         // Адрес переменной для
                              // идентификатора процесса.
);
```

Данная функция возвращает идентификатор потока. Кроме того, если ей передается указатель на DWORD в IpdwProcessIdl(), в целевой переменной возвращается идентификатор процесса.

Получение имен и дескрипторов модулей. Обычно одному процессу принадлежит много модулей, загруженных в его адресное пространство, и это, естественно, усложняет задачу получения дескрипторов и имен модулей.

Псевдодескрипторы процессов. Функция GetcurrentProcess возвращает псевдодескриптор текущего процесса:

HANDLE GetCurrentProcess(VOID) ;

Псевдодескриптор (pseudohandle) представляет собой упрощенный вариант дескриптора. По определению, псевдодескриптор – это зависимое от процесса число, которое служит идентификатором процесса и может использоваться в вызовах тех API-функций, которым требуемся дескриптор процесса.

Хотя назначение псевдодескрипторов и обычных дескрипторов почти одно и то же, у них все же есть некоторые существенные различия. Псевдодескрипторы не могут наследоваться порожденными процессами, как настоящие дескрипторы (real handler). К тому же псевдодескрипторы ссылаются только на текущий процесс, а настоящие дескрипторы могут ссылаться и на внешний (foreign).

Windows предоставляет возможность получения настоящего дескриптора по псевдодескриптору при помощи API-функции DuplicateHandle. Она определяется как

```
BOOL DuplicateHandle(
HANDLE hSourceProcessHandle,  // Дескриптор процесса-источника.
HANDLE hSourceHandle,         // Копируемый дескриптор.
HANDLE hTargetProcessHandle,  // Дескриптор процесса-приемника.
```

LPHANDLE IpTargetHandle, // Указатель на копию дескриптора.
DWORD dwDesiredAccess, // Доступ к копии дескриптора.
BOOL bInheritHandle, // Флаг наследования дескриптора.
DWORD dwOptions // Необязательные опции.
);

Перечисление процессов в Windows 9x (использование ToolHelp32)

ToolHelp32 – это семейство функций и процедур, составляющих подмножество Win32 API, которые позволяют получить сведения о некоторых низкоуровневых аспектах работы ОС, В частности, сюда входят функции, с помощью которых можно получить информацию обо всех процессах, выполняющихся в системе в данный момент, а также потоках, модулях, принадлежащих каждому процессу. Большинство данных, получаемых от функций ToolHelp32, используется, главным образом, приложениями, которые должны заглядывать "внутрь" ОС.

Семейство процедур и функций ToolHelp32 API доступно только в варианте реализации Win32 для Windows 95/98 и др. В среде Windows NT вызов их приведет к нарушению системы защиты и безопасности NT-процессов.

Типы и определения функций ToolHe1p32 размещаются в модуле TlHelp32, поэтому при работе с этими функциями не забудьте включить его имя в список инструкции uses (в Delphi).

Моментальные снимки. Благодаря многозадачной природе среды Win32 такие объекты, как процессы, потоки, модули и т.п., постоянно создаются, разрушаются и модифицируются. И поскольку состояние компьютера непрерывно изменяется, системная информация, которая, возможно, будет иметь значение в данный момент, через секунду уже никого не заинтересует. Например, предположим, что необходимо написать программу для регистрации всех модулей, загруженных в систему. Поскольку операционная система в любое время может прервать выполнение потока, отрабатывающего программу, чтобы предоставить какие-то кванты времени другому потоку в системе, модули теоретически могут создаваться и разрушаться даже в момент выборки информации о них.

В этой динамической среде имело бы смысл на мгновение заморозить систему, чтобы получить такую системную информацию. В ToolHe1p32 не предусмотрено средств замораживания системы, но есть функция, с помощью которой можно сделать "снимок" системы в заданный момент времени. Эта функция называется CreateToolhelp32Snapshot(), и ее объявление (в Delphi) выглядит следующим образом:

function CreateToolhelp32Snapshot(dwFlags, th32ProcesslD: DWORD): THandle; stdcall;

Параметр dwFlags означает тип информации, подлежащий включению в моментальный снимок. Этот параметр может иметь одно из перечисленных в табл. 2 значений.

2. Значения параметра dwFlags

Значение	Описание
TH32CS_INHERIT	Означает, что дескриптор снимка будет наследуемым
TH32CS_SNAPALL	Эквивалентно заданию значений TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS И TH32CS_SNAPTHREAD
TH32CS_SNAPHEAPLIST	Включает в снимок список куч заданного процесса Win32
TH32CS_SNAPMODULE	Включает в снимок список модулей заданного процесса Win32
TH32CS_SNAPPROCESS	Включает в снимок список процессов Win32
TH32CS_ SNAPTHREAD	Включает в снимок список потоков Win32

Функция CreateToolhelp32Snapshot() возвращает дескриптор созданного снимка или -1 в случае ошибки. Возвращаемый дескриптор работает подобно другим дескрипторам Win32 относительно процессов и потоков, для которых он действителен.

По завершении работы с созданным функцией CreateToolhelp32 Snapshot() дескриптором, для освобождения связанных с ним ресурсов используйте функцию CloseHandle().

Обработка информации о процессах. Имея дескриптор снимка, содержащий информацию о процессах, можно воспользоваться двумя функциями ToolHelp32, которые позволяют последовательно просмотреть сведения обо всех процессах в системе. Функции Process32First() и Process32Next() определены следующим образом:
function Process32First(hSnapshot: THandle; var Ippe: TProcessEntry32): BOOL; stdcall;
function Process32Next(hSnapshot: THandle; var Ippe: TProcessEntry32): BOOL; stdcall;

Первый параметр у обеих функций является дескриптором снимка, возвращаемым функцией CreateToolhelp32Snapshot().

Второй параметр, Ippe, представляет собой запись TProcessEntry32, которая передается по ссылке. По мере прохождения по элементам перечисления функции будут заполнять эту запись информацией о следующем процессе. Запись TProcessEntry32 определяется так:

```
type TProcessEntry32 = record
dwSize: DWORD;
cntUsage: DWORD;
th32ProcessID: DWORD;
th32DefaultHeapID: DWORD;
th32ModuleID: DWORD;
cntThreads: DWORD;
th32ParentProcessID: DWORD;
pcPriClassBase: Longint;
dwFlags; DWORD;
szExeFile: array[Q..MAX_PATH – 1] of Char;
end;
```

- dwSize – размер записи TProcessEntry32. До использования этой записи поле dwSize должно быть инициализировано значением SizeOf (TProcessEntry32);
- cntUsage – значение счетчика ссылок процесса. Когда это значение станет равным нулю, операционная система выгрузит процесс;
- th32ProcessID – идентификационный номер процесса.
- th32DefaultHeapID – идентификатор {ID} для кучи процесса, действующей по умолчанию. Этот ID имеет значение только для функций ToolHelp32, и его нельзя использовать с другими функциями Win32;
- thHodulelD – идентифицирует модуль, связанный с процессом. Это поле имеет значение только для функций ToolHelp32;
- cntThreads – количество потоков начало выполняться в данном процессе;
- th32ParentProcessID – идентифицирует родительский процесс для данного процесса;
- pcPriClassBase – базовый приоритет процесса. Операционная система использует это значение для управления работой потоков;
- dwFlags – зарезервировано.

В поле szExeFile содержится строка с ограничивающим нуль-символом, которая представляет собой путь и имя файла EXE-программы или драйвера, связанного с данным процессом.

После создания снимка, содержащего информацию о процессах, для опроса данных по каждому процессу следует вызвать сначала функцию Process32First(). а затем вызывать функцию Process32Next() до тех пор, пока она не вернет значение False.

Для отображения значка вместе с именем приложения, что придает программе более профессиональный вид, можно использовать API-функцию ExtractIcon из модуля ShellAPI.

Обработка информации о потоках. Для составления списка потоков некоторого процесса в ToolHelp32 предусмотрены две функции, которые аналогичны функциям, предназначенным для регистрации процессов: Thread32First() и Thread32Next (), и объявляются следующим образом;

```
function Thread32First(hSnapshot: THandle; var Ipte: TThreadEntry32): BOOL; stdcall;
```

Помимо обычного параметра hSnapshot, этим функциям также передается по ссылке параметр типа TThreadEntry32. Как и в случае функций, работающих с процессами, каждая из них заполняет запись TThreadEntry32, объявление которой имеет вид:

```
type
TThreadEntry32 = record
dwSize: DWORD;
cntUsage: DWORD;
th32ThreadID: DWORD;
th32OwnerprocessiD: Dword;
tpBasePri: Longint;
tpDeltaPri: Longint;
dwFlags: DWORD;
end;
```

- dwSize – размер записи, и поэтому оно должно быть инициализировано значением SizeOf (TThreadEntry32) до использования этой записи;
- cntUsage – счетчик ссылок данного потока. При обнулении этого счетчика поток выгружается операционной системой;
- th32ThreadID – идентификационный номер потока, который имеет значение только для функций ToolHelp32;
- th320wnerProcessID – идентификатор (ID) процесса, которому принадлежит данный поток. Этот ID можно использовать с другими функциями Win32;
- tpBasePri – базовый класс приоритета потока. Это значение одинаково для всех потоков данного процесса. Возможные значения этого поля обычно лежат в диапазоне 4...24. Описания этих значений приведены в табл. 3.

3. Значения приоритетов потоков

Значение	Описание
4	Ожидающий
8	Нормальный
13	Высокий
24	Реальное время

- tpDeltaPri – дельта-приоритет (разницу), определяющий величину отличия реального приоритета от значения tpBasePri. Это число со знаком, которое в сочетании с базовым классом приоритета отображает общий приоритет потока. Константы, определенные для всех возможных значений дельта-приоритета, перечислены в табл. 4.

4. Значения констант дельта-приоритетов

Константа	Значение
THREAD_PRIORITY_IDLE	–15
THREAD_PRIOR ITY_LOWEST	–2
THREAD PRIORITY BELOW NORMAL	–1
THREAD_PRIOR IT Y_NORMAL	0
THREAD_PRIOR I TY_ABOVE_NORHAL	1
THREAD_PRIOR ITY_H IGHEST	2
THREAD_PRI OR ITY_TIME_CRIT ICAL	15

- dwFlags – в данный момент зарезервировано и не должно использоваться.

Списки потоков, полученные с помощью функций ToolHelp32 не связываются с определенным потоком. Поэтому при сканировании потоков нужно обязательно проверять результат так, чтобы потоки были связаны с интересующим вас потоком.

Обработка информации о модулях. Опрос модулей выполняется практически так же, как опрос процессов или потоков. Для этого в ToolHe1p32 предусмотрены две функции: Module32First() и Module32Next(), которые определяются следующим образом:

function Module32First(hSnapshot: THandle; var Ipme: TModuleEtitry32): BOOL; stdcall;
function Module32Next(hSnapshot: THandle; var Ipme: TModuleEntry32): BOOL; stdcall;

ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Составить следующую программу, которая:

1) принимая дескриптор, имя или полное имя модуля, возвращает другие два элемента в своих выходных параметрах (выполнить задание для своей программы и для любой известной библиотеки);
2) будет выполнять последовательно по шагам следующее:
   а) используя функцию GetCurrentProcessId, определит идентификатор текущего процесса;
   б) используя функцию GetCurrentProcess, определит псевдодескриптор текущего процесса;
   в) используя функцию DuplicateHandle и значение псевдодескриптора, определит дескриптора текущего процесса;
   г) используя функцию OpenProcess, определит копию дескриптора текущего процесса;
   д) закроет дескриптор, полученный функцией DuplicateHandle;
   ж) закроет дескриптор, полученный функцией OpenProcess;
   3) Выдает список перечисления всех процессов, потоков, модулей и их свойства в системе.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое процесс и поток? Что включает в себя понятие поток?
2. Понятие моментального снимка? С помощью какой функции его можно получить?
3. Назовите функцию получения идентификатора.
4. Что такое дескриптор процесса? Чем он отличается от псевдодескриптора?
5. Понятие мультипрограммирования, критерии эффективности.

ПОТОКИ

Цель работы: получение практических навыков по использованию Win32 API для исследования потоков

ОСНОВНЫЕ ПОЛОЖЕНИЯ

Дескрипторы и идентификаторы потоков

Функция CreateProcess возвращает идентификатор и дескриптор первого (и только первого) потока, выполняющегося во вновь созданном процессе. Функция Create Thread тоже возвращает идентификатор потока, но область действия которого – вся система.

Поток может использовать функцию GetCurrentThreadId, чтобы получить собственный ID. Функция GetWindowThreadProcessId возвращает идентификатор того потока, который создал конкретное окно.

Согласно документации Win32, Win32 API не предлагает способа для получения дескриптора потока по его идентификатору. Если бы дескрипторы можно было находить таким образом, то процесс, которому принадлежат потоки, завершался бы неудачей, так как другой процесс смог бы выполнять несанкционированные операции с его потоками, например, приостанавливать поток, возобновлять его действие, изменять приоритет или завершать его работу. Запрашивать дескриптор следует у процесса, создавшего данный поток, или у самого потока.

Наконец, поток может вызывать функцию GetCurrentThread для получения собственного псевдодескриптора. Как и в случае псевдодескрипторов процессов, псевдодескриптор потока может использоваться только для вызова процесса и не может наследоваться. Можно использовать функцию Duplicate Handle для получения настоящего дескриптора потока по его псевдодескриптору так же, как это делается в процессах.

Приоритет потоков

Термин многозадачность или мультипрограммирование обозначает возможность управлять несколькими процессами (или несколькими потоками) на базе одного процессора. Многопроцессорной обработкой называется управление некоторым числом процессов или потоков на нескольких процессорах.

Компьютер может одновременно выполнять команды двух разных процессов только в многопроцессорной среде. Однако даже на одном процессоре с помощью переключения задач можно создать впечатление, что одновременно выполняются команды нескольких процессов.

В старой, 16-разрядной, Windows существовал только один поток. Более того, в данной системе был реализован метод кооперативной многозадачности, который состоит в том, что каждое приложение само отвечает за высвобождение единственного системного потока, после чего могут выполняться другие приложения. Если программа выполняла задачи, требующие значительного времени, такие как форматирование гибкого диска, все другие загруженные приложения должны были ждать.

Уровни приоритета потоков. Win32 значительно отличается от Win16. Во-первых, она является многопоточной (multithreaded), что определяет ее многозадачность. Во-вторых, в ней реализована модель вытесняющей многозадачности, в которой операционная система решает, когда каждый поток получает процессорное время, выделяемое квантами времени, и сколько именно времени выделяется. Временной интервал в Windows называется квантом.

Продолжительность кванта времени зависит от аппаратуры и может фактически меняться от потока к потоку. Например, базовое значение в Windows 95 составляет 20 мс, для Windows NT Workstation (на базе процессора Pentium) – 30 мс, для Windows NT Server – 180 мс.

Рассмотрим, каким образом Windows выделяет кванты времени потокам в системе. Эти процедуры в Windows 9x и Windows NT довольно похожи, но не идентичны.

Каждый поток в системе имеет уровень приоритета, который представляет собой число в диапазоне 0...31. Необходимо отметить следующее:

1. Если существуют какие-либо потоки с приоритетом 31, которые требуют процессорного времени, т.е. не находятся в состоянии ожидания (not idle), операционная система перебирает эти потоки (независимо от того, каким процессам они принадлежат), выделяя по очереди каждому из них кванты времени. Потокам с более низким приоритетом кванты времени со всем не выделяются, поэтому они не выполняются. Если нет активных потоков с приоритетом 31, операционная система ищет активные потоки с уровнем приоритета 30 и т.д. Не следует забывать, однако, что потоки довольно часто простаивают. Тот факт, что приложение загружено, не означает, что все его потоки активны. Поэтому у потоков с более низким приоритетом все же есть возможность работать. Более того, если пользователь нажимает клавишу, относящуюся к процессу, потоки которого простаивают, операционная система временно выделяет процессорное время соответствующему потоку, чтобы он мог обработать нажатие клавиши.
2. Приоритет 0 зарезервирован исключительно за специальным системным потоком, который называется потоком нулевой страницы (zero page thread). Он освобождает незадействованные области памяти. Существует также поток idle, который работает с уровнем приоритета 0, опрашивая систему в поисках какой-нибудь работы.
3. Если поток с произвольным приоритетом выполняется в тот момент, когда потоку с большим приоритетом потребовалось процессорное время (например, он получает сообщение о том, что пользователь щелкнул мышью), операционная система немедленно вытесняет поток с меньшим приоритетом и отдает процессорное время потоку с большим. Таким образом, поток может не успеть завершить выделенный ему квант времени.
4. Для того чтобы перейти с одного потока на другой, система осуществляет переключение контекста (context switch). Это процедура, сохраняющая состояние процессора (регистров и стека) и загрузки соответствующих значений другого потока.

Назначение приоритета потоку. Назначение потоку приоритета происходит в два этапа. Во-первых, каждому процессу в момент создания присваивается класс приоритета. Узнать класс приоритета можно с помощью функции GetPriorityClass, а изменить – с помощью функции SetPriorityClass. В таблице 5 приведены имена классов приоритета процессов, уровни приоритета и константы, которые используются с этими вышеупомянутыми функциями (как и с функцией CreateProcess).

5. Соответствие констант уровням приоритетов

Имя класса приоритета	Уровень приоритета класса	Символьная константа
Idle	4	*IDLE_PRIORITY_CLASS=&H40*
Normal	8	*NORMAL_PRIORITY_CLASS=&H20*
High	13	*HIGH_ PRIORITY_CLASS=&H80*
Realtime	24	*REALTIME_ PRIORITY_CLASS=&H100*

Большинство процессов должно получать класс уровня приоритета Normal (обычный). Однако некоторым приложениям, таким как приложения мониторинга системы, возможно, более уместно назначать приоритет Idle (ожидания). Назначения приоритета *Realtime* (реального времени) обычно следует избегать, потому что в этом случае потоки изначально получают приоритет более высокий, чем системные потоки, такие как потоки ввода от клавиатуры и мыши, очистки кэша и обработки нажатия клавиш Ctrl+Alt+Del. Такой приоритет может быть подходящим для краткосрочных, критичных к времени выполнения процессов, которые относятся к взаимодействию с аппаратурой.

При создании уровень приоритета потока по умолчанию устанавливается равным уровню класса приоритета процесса, создавшего данный поток. Тем не менее, можно использовать функцию SetThreadPriority, чтобы изменить приоритет потока:

```
BOOL SetThreadPriority (
HANDLE hThread,                 // Дескриптор потока.
int nPriority                   // Уровень приоритета потока.
);
```

Параметр nPriority используется для изменения приоритета потока относительно приоритета процесса, которому принадлежит данный поток.

Повышение приоритета потока и квант изменений приоритета. Диапазон приоритета 1...15 известен как диапазон динамического приоритета, а диапазон 16...31 – как диапазон приоритета реального времени.

В Windows NT приоритет потока, находящийся в динамическом диапазоне, может временно повышаться операционной системой в различные моменты времени Соответственно, нижний уровень приоритета потока (установленный программистом с помощью API функции) называется уровнем его базового приоритета API функция Windows NT SetProcessPriority-Boost может использоваться для разрешения или запрещения временных изменений приоритета. Правда, она не поддерживается в Windows 9x.

Бывают случаи, когда кванты времени, выделяемые потоку, временно увеличиваются.

Стремясь плавно выполнять операции, Windows будет повышать приоритет потока или увеличивать продолжительность его кванта времени при следующих условиях:

1) если поток принадлежит приоритетному процессу, т.е. процессу, окно которого активно и имеет фокус ввода;
2) если поток первым вошел в состояние ожидания;
3) если поток выходит из состояния ожидания;
4) если поток совсем не получает процессорного времени.

Состояния потоков

Потоки могут находиться в одном из нескольких состояний:

1) Ready (готов) – находящийся в пуле потоков, ожидающих выполнения;
2) Running (выполнение) – выполняющийся на процессоре;
3) Waiting (ожидание), также называется idle или suspended, приостановленный – в состоянии ожидания, которое завершается тем, что поток начинает выполняться (состояние Running) или переходит в состояние Ready;
4) Terminated (завершение) – завершено выполнение всех команд потока. Впоследствии его можно удалить. Если поток не удален, система может вновь установить его в исходное состояние для последующего использования.

Синхронизация потоков

Выполняющимся потокам часто необходимо каким-то образом взаимодействовать. Например, если несколько потоков пытаются получить доступ к некоторым глобальным данным, то каждому потоку нужно предохранять данные от изменения другим потоком. Иногда одному потоку нужно получить информацию о том, когда другой поток завершит выполнение задачи. Такое взаимодействие обязательно между потоками как одного, так и разных процессов.

Синхронизация потоков (thread synchronization) – это обобщенный термин, относящийся к процессу взаимодействия и взаимосвязи потоков. Синхронизация потоков требует привлечения в качестве посредника самой операционной системы. Потоки не могут взаимодействовать друг с другом без ее участия.

В Win32 существует несколько методов синхронизации потоков. В зависимости от конкретной ситуаций один метод более предпочтителен, чем другой. Рассмотрим эти методы.

Критические секции. Один из методов синхронизации потоков состоит в использовании критических секций (critical sections). Это единственный метод синхронизации потоков, который не требует привлечения ядра Windows. (Критическая

секция не является объектом ядра). Однако этот метод может использоваться только для синхронизации потоков одного процесса.

Критическая секция – это некоторый участок кода, который в каждый момент времени может выполняться только одним из потоков. Если код, используемый для инициализации массива, поместить в критическую секцию, то другие потоки не смогут войти в этот участок кода до тех пор, пока первый поток не завершит его выполнение.

До использования критической секции необходимо инициализировать ее с помощью процедуры Win32 API InitializeCriticalSection(), которая определяется (в Delphi) следующим образом:

```
procedure InitializeCriticalSection(var IpCriticalSection:
TRTLCriticalSection); stdcall;
```

Параметр IpCriticalSection представляет собой запись типа TRTLCriticalSection, которая передается по ссылке.

После заполнения записи в программе можно создать критическую секцию, поместив некоторый участок ее текста между вызовами функций EnterCriticalSection() и LeaveCriticalSection(). Эти процедуры определяются следующим образом:

```
procedure EnterCriticalSection(var IpCriticalSection:
TRTLCriticalSection); stdcall;
procedure LeaveCriticalSection(var IpCriticalSection:
TRTLCriticalSection); stdcall;
```

Параметр IpCriticalSection, который передается этим процедурам, является не чем иным, как записью, созданной процедурой InitializeCritical- Section().

Функция EnterCriticalSection проверяет, не выполняет ли уже какой-нибудь другой поток критическую секцию своей программы, связанную с данным объектом критической секции. Если нет, поток получает разрешение на выполнение своего критического кода, точнее, ему не запрещают это делать. Если да, то поток, обратившийся с запросом, переводится в состояние ожидания, а о запросе делается запись. Так как нужно создавать записи, объект "критическая секция" представляет собой структуру данных.

Когда функция LeaveCriticalSection вызывается потоком, который владеет в текущий момент разрешением на выполнение своей критической секции кода, связанной с данным объектом "критическая секция", система может проверить, нет ли в очереди другого потока, ожидающего освобождения этого объекта. Затем система может вынести ждущий поток из состояния ожидания, и он продолжит свою работу (в выделенные ему кванты времени).

По окончании работы с записью TRTLCriticalSection необходимо освободить ее, вызвав процедуру DeleteCriticalSection(), которая определяется следующим образом:

```
procedure DeleteCriticalSection(var IpCriticalSection:
TRTLCriticalSection); stdcall;
```

Синхронизация с использованием объектов ядра. Многие объекты ядра, включая процесс, поток, файл, мьютекс, семафор, уведомление об изменении файла и событие, могут находиться в одном из двух состояний – "свободно" (*signaled*) и "занято" (*nonsignaled*). Вероятно, проще представлять себе эти объекты подключенными к лампочке. Если свет горит, объект свободен, в обратном случае объект занят.

Например, в момент создания процесса его объект ядра находится в состоянии "занято". Когда процесс завершается, объект переходит в состояние "свободно". Аналогично выполняющиеся потоки (т.е. их объекты) пребывают в состоянии "занято", но переходят в состояние "свободно", когда завершают работу. На самом деле некоторые объекты, такие как мьютекс, семафор, событие, уведомление об изменении файла, таймер ожидания, существуют исключительно для того, чтобы вырабатывать сигналы "свободно" и "занято".

Смысл всей этой "сигнализации" в том, чтобы поток мог приостанавливать свою работу до того момента, когда заданный объект перейдет в состояние "свободно". Например, поток одного процесса может временно прекратить работу до завершения другого, просто подождав, когда объект ядра этого другого процесса перейдет в состояние "свободно".

Посредством вызова функций WaitForSingleObject и WaitForMultipleObjects поток приостанавливает свое выполнение до того момента, когда заданный объект (или объекты) перейдет в состояние "свободно". Рассмотрим функции WaitForSingleObject, декларация которой выглядит так:

```
DWORD WaitForSingleObject(
HANDLE hHandle,                 // Дескриптор объекта ожидания.
DWORD dwMilliseconds            // Время ожидания в миллисекундах.
);
```

Параметр hHandle является дескриптором объекта, уведомление о свободном состоянии которого требуется получить, а dwMilliseconds – это время, которое вызывающий поток готов ждать. Если dwMilliseconds равно нулю, функция немедленно вернет текущий статус заданного объекта. Таким образом, можно протестировать состояние объекта. Параметру можно также присваивать значение символьной константы *INFINITE (= -1),* в этом случае вызывающий поток будет ждать неограниченное время.

Функция WaitForSingleObject переводит вызывающий поток в состояние ожидания до того момента, когда она передаст ему свое возвращаемое значение, например:

WAIT_OBJECT_0 – объект находится в состоянии "свободно";

WAIT_TIMEOUT – интервал ожидания, заданный dwMilliseconds, истек, а нужный объект по прежнему находится в состоянии "занято";

WAIT_ABANDONED относится только к мьютексу и означает, что объект не был освобожден потоком, который владел им до своего завершения;

WAIT_FAILED – при выполнении функции произошла ошибка.

- М ь ю т е к с (MUTual Exclusions – взаимоисключения) – это объект ядра, который можно использовать для синхронизации потоков из разных процессов. Он может принадлежать или не принадлежать некоторому потоку. Если мьютекс принадлежит потоку, то он находится в состоянии "занято". Если данный объект не относится ни к одному потоку, то он находится в состоянии "свободно". Другими словами, принадлежать для него означает быть в состоянии "занято".

Если мьютекс не принадлежит ни одному потоку, первый поток, который вызовет функцию WaitForSingleObject, завладевает данным объектом, и тот переходит в состояние "занято". В определенном смысле мьютекс похож на выключатель, которым может пользоваться любой поток по принципу "первым пришел – первым обслужили" (first-come-first,-served).

Дело в том, что при попытке с помощью вызова функции WaitForSingleObject завладеть мьютексом, который уже находится в состоянии "занято", поток переводится в состояние ожидания до того момента, когда данный объект освободится, т.е. когда "владелец" мьютекса его освободит (переведет в состояние "свободно").

По принципу своего действия мьютексы очень похожи на критические секции, за исключением двух моментов: во-первых, мьютексы можно использовать для синхронизации потоков, переступая через границы процессов; во-вторых, мьютексу можно присвоить имя и путем ссылки на это имя создать дополнительные дескрипторы существующих объектов мьютексов.

Мьютексы создаются с помощью вызова функции CreateMutex:

```
HANDLE CreateMutexi
LPSECURITY_ATTRIBUTES        // Указатель на атрибуты защиты.
IpMutexAttributes,
BOOL bInitialOwner,          // Флаг первоначального владельца.
LPCTSTR IpName               // Указатель на имя мьютекса.
);
```

Параметр IpMutexAttributes – это указатель на запись типа TSecurityftttributes. Обычно в качестве данного параметра передастся значение nil, и в этом случае используются атрибуты защиты, действующие по умолчанию.

Параметр blnitialOwner определяет, следует ли считать поток, создающий мьютекс, его владельцем. Если этот параметр равен False, значит, мьютекс не имеет владельца.

Параметр IpName представляет имя мьютекса. Если нет необходимости присваивать мьютексу имя, следует установить этот параметр равным nil. Если же значение этого параметра отлично от nil, функция выполнит в системе поиск мьютекса с таким же именем. При успешном завершении поиска функция вернет дескриптор найденного мьютекса, в противном случае возвращается дескриптор нового мьютекса. При наличии имени этот объект может совместно использоваться несколькими процессами. Если каким-то процессом создается мьютекс с именем, то поток другого процесса может вызывать функции CreateMutex или OpenMutex с тем же самым именем. В любом случае система просто передаст вызывающему потоку дескриптор исходного мьютекса. Другой способ совместно использовать мьютекс – вызвать функцию DuplicateHandle.

Чтобы работать с несколькими процессами, данный объект должен быть совместно используемым. Причина проста: чтобы завладеть мьютексом или освободить его, потоку потребуется его дескриптор. Поток освобождает этот объект с помощью вызова функции ReleaseMutex:

```
BOOL ReleaseMutex(
HANDLE hMutex                // Дескриптор мьютекса.
);
```

А что случится, если владеющий мьютексом поток завершится, предварительно не освободив его? В действительности система сама освобождает такой мьютекс. Поток, который вызывает функцию WaitForSingleObject для этого объекта, получит возвращенное значение WAIT_ABANDONED, которое указывает на возникшие проблемы с только что завершимся потоком-владельцем. В этом случае ждущий поток должен определить, стоит продолжать выполнение в обычном режиме или нет.

По завершении использования мьютекса необходимо закрыть его с помощью функции Win32 API CloseHandle().

- С о б ы т и я используются в качестве сигналов о завершении какой-либо операции. Однако в отличие от мьютексов они не принадлежат ни одному потоку. Например, поток *A* создает событие с помощью функции CreateEvent и устанавливает объект в состояние "занято". Поток *B* получает дескриптор этого объекта, вызвав функцию OpenEvent, затем вызывает функцию WaitForSingleObject, чтобы приостановить работу до того момента, когда поток *A* завершит конкретную задачу и освободит указанный объект. Когда это произойдет, система выведет из состояния ожидания поток *B*, который теперь владеет информацией, что поток *A* завершил выполнение своей задачи.

Объявление функции *CreateEvent* записывается таким образом:

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES        // Указатель на атрибуты защиты.
IpEventAttributes,
BOOL bManualReset,           // Флаг интерактивного события.
BOOL bInitialState,          // Флаг первоначального состояния.
LPCTSTR IpName               // Указатель на имя события.
);
```

Функция возвращает дескриптор создаваемого объекта "событие". Первый параметр определяет, наследуется ли дескриптор порожденными процессами. Если IpEventAttributes имеет значение NULL, дескриптор наследоваться не может.

Если параметр bManualReset имеет значение TRUE, то при освобождении объект остается в этом состоянии (в отличие от объекта "мьютекс"). Это значит, что все потоки, ожидающие перехода данного объекта в состояние "свободно", будут выведены системой из состояния ожидания. Такой объект называется событием с ручным сбросом (*manual-reset event*), поскольку "разбуженный" (выведенный из состояния ожидания) поток может самостоятельно сбросить состояние объекта "событие" в "занято". Если параметр bManualReset имеет значение FALSE, то система автоматически сбрасывает состояние рас-

сматриваемого объема в "занято" после "пробуждения" первого потока, ожидающего освобождения данного объекта. Только один поток выводится из состояния ожидания, как и в случае с мьютексами. Такое событие называют событием с автоматическим сбросом (*auto-reset event*).

Параметр bInitialState определяет первоначальное состояние (если TRUE, то "свободно", если FALSE, то "занято") данного события. Параметру IpName может быть присвоено имя события. Имя предоставляет способ совместного использования, например посредством функции OpenEvent.

В качестве дополнительного варианта, если нет необходимости иметь дело с вопросами защиты, можно установить IpEventAttributes в NULL (0&). В таком случае декларация примет следующий вид:

```
Declare Function CreateEvent Lib "kernel32" Alias "CreateEventA" (
ByVal IpEventAttributes As Long, _
By Val bManualReset As Long, _
By Val blmtialState As Long, _
By Val IpName As String _
) As Long
```

Дескриптор события должен быть закрыт с использованием функции CloseHandle.

Объявление функции OpenEvent:

```
HANDLE OpenEvent(
DWORD dwDesiredAccess,          // Флаг доступа.
BOOL blnheritHandle,            // Флаг наследования.
LPCTSTR IpName                  // Указатель на имя события.
);
```

Здесь параметр dwDesiredAccess может принимать одно из трех значений:

1) EVENT_ALL_ACCESS предоставляет полный доступ к событию;

2) EVENT_MODIFY_STATE разрешает использование дескриптора события в функциях SetEvent и ResetEvent, так что вызывающий процесс может изменить состояние данного события (но ничего больше). Это важно для событий со сбросом вручную;

3) SYNCHRONIZE разрешает использование дескриптора события в любых функциях ожидания (таких как WaitForSingleObject), ждущих освобождения данного объекта.

Каждая из этих функций принимает дескриптор события в качестве аргумента. Функция SetEvent устанавливает состояние данного события в "свободно", а Reset Event "сбрасывает" событие, т.е. присваивает событию статус "занято", функция PulseEvent вызывает SetEvent для освобождения ожидающих потоков, а затем вызывает ResetEvent для перевода данного события в состояние "занято".

- С е м а ф о р ы являются одним из методов синхронизации потоков, в которых применен принцип действия мьютексов, но с добавлением одной существенной детали. В них заложена возможность подсчета ресурсов, что позволяет заранее определенному числу потоков одновременно войти в синхронизуемый участок кода. Для создания семафора используется функция CreateSemaphore(), которая объявляется следующим образом:

```
function CreateSemaphoreflpSemaphoreAttributes: PSecurityAttributes;
llnitialCoimt, iMaximumCount: Longint; IpName: PChar): THandle; stdcall;
```

Как и в случае функции CreateMutex(), первым параметром, передаваемым функции CreateSemaphore(), является указатель на запись TSecurityAttributes, причем значение Nil соответствует согласию на использование стандартных атрибутов защиты.

Параметр llnitialCount представляет собой начальное значение счетчика семафорного объекта. Это число может находиться в диапазоне от нуля до значения IMaximumCount. Семафор доступен, если значение этого параметра больше нуля. Когда поток вызывает функцию WaitForSingleObject() или любую другую, ей подобную, значение счетчика семафора уменьшается на единицу. И, наоборот, при вызове потоком функции ReleaseSemaphore() значение счетчика семафора увеличивается на единицу.

С помощью параметра IMaximumCount задается максимальное значение счетчика семафорного объекта. Если семафор используется для подсчета некоторых ресурсов, это число должно представлять общее количество доступных ресурсов.

Параметр IpName содержит имя семафора. Поведение этого параметра аналогично поведению одноименного параметра функции CreateMutex().

Функция ReleaseSemaphore (), используемая для увеличения значения счетчика семафора, имеет больше параметров, чем ее "коллега" ReleaseMutex(). Объявление функции ReleaseSemaphore() выглядит следующим образом:

```
function ReleaseSemaphore(hSemaphore: THandle; IReleaseCount:
Longint; IpPreviousCount: Pointer): BOOL; stdcall;
```

С помощью параметра IReleaseCount можно задать число, на которое будет уменьшено значение счетчика семафора. При этом старое значение счетчика будет сохранено в переменной типа Longint, на которую указывает параметр IpPreviousCount, если его значение не равно Nil. Скрытый смысл этого средства состоит в том, что семафор никогда не принадлежит ни одному отдельному потоку. Предположим, что максимальное значение счетчика семафора было равно 10, и десять потоков вызвали функцию WaitForSingleObject(). В результате счетчик потоков сбрасывается до нуля и тем самым семафор переводится в недоступное состояние. После этого достаточно одному из потоков вызвать функцию

ReleaseSemaphore() и в качестве параметра lReleaseCount передать число 10, как семафор не просто будет снова пропускать потоки, т.е. станет доступным, но и увеличит значение своего счетчика до прежнего числа – до 10. Это мощное средство может привести к возникновению в приложении трудно отслеживаемых ошибок, поэтому следует использовать его с большой осторожностью.

Семафоры могут быть полезны при совместном использовании ограниченных ресурсов. Предположим, имеется три приложения, каждое из которых должно выполнить вывод на печать, а у компьютера только два параллельных порта. Установив семафор с начальным значением счетчика ресурсов, равным двум, можно заставить приложения запрашивать сервис печати только тогда, когда есть свободный параллельный порт.

Для освобождения дескриптора семафора, выделенного ему с помощью функции CreateSemaphore(), не забудьте вызвать функцию CloseHandle().

- Ждущий таймер (waitable timer) представляет собой новый тип объектов синхронизации, поддерживаемый в Windows NT версии 4.0 и выше. Это полноценный объект синхронизации, который может использоваться для организации задержки в одном или нескольких приложениях.

Ждущий таймер работает в трех режимах. В режиме "ручного сброса" таймер переходит в установленное состояние при истечении заданной задержки и остается установленным до тех пор, пока функция SetWaitableTimer не задаст новую задержку. В режиме "автоматического сброса" таймер переходит в установленное состояние при истечении заданной задержки и остается установленным до первого успешного вызова функции ожидания. В этом режиме он напоминает объект Event в режиме автоматического сброса, поскольку каждый раз при истечении времени задержки разрешается выполнение лишь одной нити. Наконец, ждущий таймер может выполнять функции интервального таймера, который перезапускается с заданной задержкой после каждого срабатывания объекта.

Главная особенность, отличающая ждущие таймеры от системных, – то, что ждущие таймеры могут совместно использоваться несколькими приложениями. Например, можно приостановить несколько приложений в фоновом режиме так, чтобы они "просыпались" каждые несколько часов для выполнения некоторой операции.

Процессы получают дескрипторы ждущих таймеров так же, как они получают дескрипторы мьютексов: дублированием, наследованием или открытием по имени.

В таблице 6 перечислены функции, предназначенные для работы со ждущими таймерами.

6. Функции для работы со ждущими таймерами

Функция	Описание
CancelWaitableTimer	Останавливает работу ждущего таймера. Таймер остается в текущем состоянии
CreateWaitableTimer	Создает объект ждущего таймера. Если таймер с заданным именем уже существует, он открывается
OpenWaitableTirner	Открывает существующий ждущий таймер
SetWaitableTimer	Запускает ждущий таймер с заданной продолжительностью и интервалом срабатывания

ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. Учитывая особенности методов синхронизации, сформулировать три задачи, демонстрирующие возможности ОС по синхронизации потоков следующими способами:
   1) критические секции;
   2) мьютексы;
   3) события.
2. Разработать программу, реализующую решение сформулированных задач с визуализацией результатов работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие алгоритмы планирования потоков вы знаете?
2. Планирование потоков в системах реального времени.
3. Понятие синхронизации.
4. Как осуществляется синхронизация потоков критическими секциями?
5. Каковы особенности синхронизации с помощью мьютексов? Для каких задач они применяются?
6. Опишите применение событий для решения задачи синхронизации.

МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

Цель работы: изучение механизмов межпроцессного взаимодействия (InterProcess Communication) в Windows NT; получение практических навыков по использованию Win32 API для программирования механизмов IPC.

ОСНОВНЫЕ ПОЛОЖЕНИЯ

Отображение файлов (file mapping)

Механизм отображения файлов позволяет процессу трактовать содержимое файла как блок памяти в адресном пространстве этого процесса. Процесс может использовать обычные операции с указателями для того, чтобы считывать и изменять содержимое файла. Когда два или более процесса получают доступ к одинаковым отображениям файлов, то каждый процесс получает указатель на память в собственном адресном пространстве, которое он может использовать для модифицирования содержимого файла (рис. 5). Отсюда следует, что процессы также должны использовать объект синхронизации, например семафор, чтобы предотвратить разрушение данных в многозадачной среде.

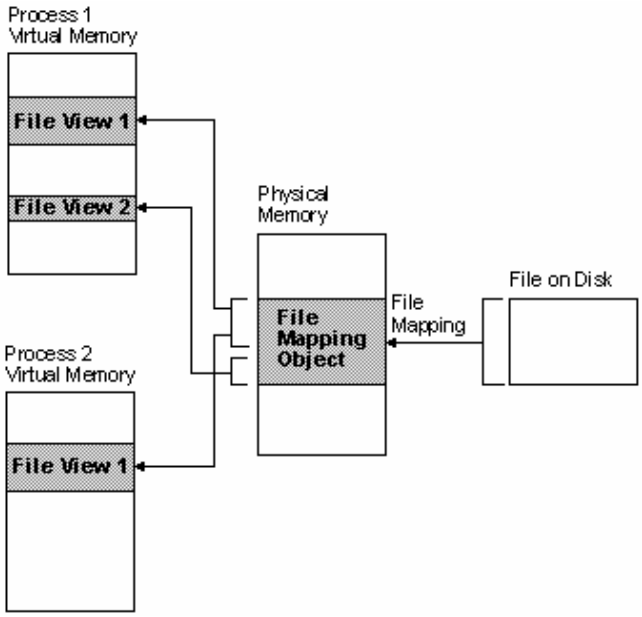

Рис. 5. Работа с файловым отображением

Программист может использовать специальный вид отображения файла для обеспечения именованной разделенной памяти между процессами. Если при создании объекта-отображения файла указывается файл подкачки системы (swapping file), то отображение файла создается как блок совместно используемой памяти. Другие процессы также могут получить доступ к этому блоку, открыв такое же отображение файла.

Отображение файлов вполне эффективно, и, кроме того, предоставляет поддерживаемые операционной системой атрибуты безопасности для того, чтобы помочь предотвратить несанкционированный доступ к данным.

Механизм отображения файлов может быть использован процессами, работающими только на локальном компьютере; он не используется для передачи данных по сети.

Рассмотрим последовательно все операции, необходимые для создания отображения файлов, чтения/записи отображений и корректного завершения работы.

1. Создание именованной совместно используемой памяти. Первый процесс вызывает функцию CreateFileMapping для того, чтобы создать объект-отображение файла с именем MyFileMappingObject. Используя флаг PAGE_READWRITE, процесс назначает права на чтение и запись данных при работе с файловым отображением.

Функция CreateFileMapping создает или открывает именованный или неименованный объект-отображение файла (подробно рассмотрена в лабораторной работе 2).

Затем процесс использует дескриптор hMapFile при вызове функции MapViewOfFile, чтобы создать представление содержимого файла в адресном пространстве процесса. Функция возвращает указатель на представление файла в памяти.

Второй процесс вызывает функцию OpenFileMapping с именем MyFileMappingObject, чтобы использовать тот же объект-отображение файла, что и первый процесс. Также как и первый процесс, второй использует функцию MapViewOfFile для сопоставления области памяти процесса отображаемому файлу.

2. Чтение/запись отображенных данных. Для чтения из представления файла в памяти разыменуем указатель, полученный с помощью функции MapViewOfFile:

```
DWORD dwLength;
dwLength = *((LPDWORD) lpMapAddress);
```

Тот же указатель используется и для записи данных в отображенный файл:

```
*((LPDWORD) lpMapAddress) = dwLength;
```

Функция FlushViewOfFile копирует указанное количество байт представления файла в памяти в физический файл, не ожидая пока произойдет операция кэшированной записи:

3. Завершение работы с отображениями. Каждый процесс вызывает функцию UnmapViewOfFile, чтобы сделать недействительным указатель на отображенную память. Этим уничтожается сопоставление адресного пространства процесса объекту-отображению файла. Если это необходимо, функция UnmapViewOfFile также копирует измененные страницы памяти на диск.

Когда все процессы завершат использование объекта-отображения файла (вызвав предыдущую функцию), они должны закрыть дескрипторы объектов-отображений с помощью функции CloseHandle:

CloseHandle(hMapFile);

Почтовые ящики (mailslot)

Почтовые ящики обеспечивают только однонаправленные соединения. Каждый процесс, который создает почтовый ящик, является "сервером почтовых ящиков" (mailslot server). Другие процессы, называемые "клиентами почтовых ящиков" (mailslot clients), посылают сообщения серверу, записывая их в почтовый ящик. Входящие сообщения всегда дописываются в почтовый ящик и сохраняются до тех пор, пока сервер их не прочтет. Каждый процесс может одновременно быть и сервером, и клиентом почтовых ящиков, создавая, таким образом, двунаправленные коммуникации между процессами.

Клиент может посылать сообщения на почтовый ящик, расположенный на том же компьютере, на компьютере в сети, или на все почтовые ящики с одним именем всем компьютерам выбранного домена. При этом широковещательное сообщение, транслируемое по домену, не может быть более 400 байт. В остальных случаях размер сообщения ограничивается только при создании почтового ящика сервером.

Почтовые ящики предлагают легкий путь для обмена короткими сообщениями, позволяя при этом вести передачу и по локальной сети, в том числе и по всему домену.

Mailslot является псевдофайлом, находящимся в памяти, и следует использовать стандартные функции для работы с файлами, чтобы получить доступ к нему. Данные в почтовом ящике могут быть в любой форме – их интерпретацией занимается прикладная программа, но их общий объем не должен превышать 64 Кб. Однако, в отличие от дисковых файлов, mailslot'ы являются временными – когда все дескрипторы почтового ящика закрыты, он и все его данные удаляются. Заметим, что все почтовые ящики являются локальными по отношению к создавшему их процессу; процесс не может создать удаленный mailslot.

Сообщения меньше чем 425 байт передаются с использованием дейтаграмм. Сообщения, больше чем 426 байт, используют передачу с установлением логического соединения на основе SMB-сеансов. Передачи с установлением соединения допускают только индивидуальную передачу от одного клиента к одному серверу. Следовательно, теряется возможность широковещательной трансляции сообщений от одного клиента ко многим серверам. Windows не поддерживает сообщения размером в 425 или 426 байт.

Когда процесс создает почтовый ящик, имя последнего должно иметь следующую форму:

\\.\mailslot\[path]name

Например:

\\.\mailslot\taxes\bobs_comments
\\.\mailslot\taxes\petes_comments
\\.\mailslot\taxes\sues_comments

Если необходимо отправить сообщение в почтовый ящик на удаленный компьютер, то следует воспользоваться NETBIOS-именем:

\\ComputerName\mailslot\[path]name

Чтобы передать сообщение всем mailslot'ам с указанным именем внутри домена, понадобится NETBIOS-имя домена:

\\DomainName\mailslot\[path]name

Для главного домена операционной системы (домен, в котором находится рабочая станция):

*\mailslot\[path]name

Клиенты и серверы, использующие почтовые ящики, при работе с ними должны пользоваться функциями, представленными в табл. 7.

7. Функции почтовых ящиков

Функция	Описание
	Серверов
CreateMailslot	Создает почтовый ящик и возвращает его дескриптор
GetMailslotInfo	Извлекает максимальный размер сообщения, размер почтового ящика, размер следующего сообщения в ящике, количество сообщений и время ожидания сообщения при выполнении операции чтения
SetMailslotInfo	Изменение таймаута при чтении из почтового ящика
DuplicateHandle	Дублирование дескриптора почтового ящика
ReadFile, ReadFileEx	Считывание сообщений из почтового ящика
GetFileTime	Получение даты и времени создания mailslot'a
SetFileTime	Установка даты и времени создания mailslot'a
GetHandleInformation	Получение свойств дескриптора почтового ящика
SetHandleInformation	Установка свойств дескриптора почтового ящика
	Клиентов
CloseHandle	Закрывает дескриптор почтового ящика для клиентского процесса
CreateFile	Создает дескриптор почтового ящика для клиентского процесса
DuplicateHandle	Дублирование дескриптора почтового ящика
WriteFile, WriteFileEx	Запись сообщений в почтовый ящик

Рассмотрим последовательно все операции, необходимые для корректной работы с почтовыми ящиками.

1. Создание почтового ящика. Операция выполняется процессом сервера с использованием функции CreateMailslot:

```
HANDLE CreateMailslot(
LPCTSTR lpName,                 // Имя почтового ящика.
DWORD nMaxMessageSize,          // Максимальный размер сообщения.
DWORD lReadTimeout,             // Таймаут операции чтения.
LPSECURITY_ATTRIBUTES           // Опции наследования и
lpSecurityAttributes            // безопасности.
);
```

2. Запись сообщений в почтовый ящик производится аналогично записи в стандартный дисковый файл. Следующий код иллюстрирует, как с помощью функций CreateFile, WriteFile и CloseHandle можно поместить сообщение в почтовый ящик:

```
LPSTR lpszMessage = "Сообщение для sample_mailslot в текущем домене.";
BOOL fResult;
HANDLE hFile;
DWORD cbWritten;

// С помощью функции CreateFile клиент открывает mailslot для записи сообщений
hFile = CreateFile("\\\\*\\mailslot\\sample_mailslot",
    GENERIC_WRITE,
    FILE_SHARE_READ, // Требуется для записи в mailslot
    (LPSECURITY_ATTRIBUTES) NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    (HANDLE) NULL);

if (hFile == INVALID_HANDLE_VALUE)
```

```
{
    ErrorHandler(hwnd, "Ошибка открытия почтового ящика");
    return FALSE;
}

// Запись сообщения в почтовый ящик
fResult = WriteFile(hFile,
    lpszMessage,
    (DWORD) lstrlen(lpszMessage) + 1, // включая признак конца строки
    &cbWritten,
    (LPOVERLAPPED) NULL);

if (!fResult)
{
    ErrorHandler(hwnd, "Ошибка при записи сообщения");
    return FALSE;
}

TextOut(hdc, 10, 10, "Сообщение отправлено успешно.", 21);

fResult = CloseHandle(hFile);

if (!fResult)
{
    ErrorHandler(hwnd, "Ошибка при закрытии дескриптора");
    return FALSE;
}

TextOut(hdc, 10, 30, "Дескриптор закрыт успешно.", 23);
return TRUE;
```

3. Чтение сообщений из почтового ящика. Создавший почтовый ящик процесс получает право считывания сообщений, из него используя дескриптор mailslot'а в вызове функции ReadFile.

Почтовый ящик существует до тех пор, пока не вызвана функция CloseHandle на сервере или пока существует сам процесс сервера. В обоих случаях все непрочитанные сообщения удаляются из почтового ящика, уничтожаются все клиентские дескрипторы и mailslot удаляется из памяти.

Функция считывает параметры почтового ящика:

```
BOOL GetMailslotInfo(
HANDLE hMailslot,                   // Дескриптор почтового ящика.
LPDWORD lpMaxMessageSize,           // Максимальный размер
                                    // сообщения.
LPDWORD lpNextSize,                 // Размер следующего
                                    // непрочитанного сообщения.
LPDWORD lpMessageCount,             // Количество сообщений.
LPDWORD lpReadTimeout               // Таймаут операции чтения.
);
```

Функция устанавливает таймаут операции чтения:

```
BOOL SetMailslotInfo(
HANDLE hMailslot,                   // Дескриптор почтового ящика.
DWORD lReadTimeout                  // Новый таймаут операции
                                    // чтения.
);
```

Каналы (pipe)

Существует два способа организовать двунаправленное соединение с помощью каналов: безымянные и именованные каналы.

1. Безымянные (или анонимные) каналы позволяют связанным процессам передавать информацию друг другу. Обычно, безымянные каналы используются для перенаправления стандартного ввода/вывода дочернего процесса так, чтобы он мог обмениваться данными с родительским процессом. Чтобы производить обмен данными в обоих направлениях, следует создать два безымянных канала. Родительский процесс записывает данные в первый канал, используя его дескриптор записи, в то время как дочерний процесс считывает данные из канала, используя дескриптор чтения. Аналогично, дочерний процесс записывает данные во второй канал и родительский процесс считывает из него данные. Безымянные каналы не могут быть использованы для передачи данных по сети и для обмена между несвязанными процессами!

2. И м е н о в а н н ы е каналы используются для передачи данных между независимыми процессами или между процессами, работающими на разных компьютерах. Обычно, процесс сервера именованных каналов создает именованный канал с известным именем или с именем, которое будет передано клиентам. Процесс клиента именованных каналов, зная имя созданного канала, открывает его на своей стороне с учетом ограничений, указанных процессом сервера. После этого между сервером и клиентом создается соединение, по которому может производиться обмен данными в обоих направлениях. В дальнейшем наибольший интерес будут представлять именованные каналы.

При создании и получении доступа к существующему каналу необходимо придерживаться следующего стандарта имен каналов:

\\.\pipe*pipename*

Если канал находится на удаленном компьютере, то потребуется NETBIOS-имя компьютера:

ComputerName\pipe*pipename*

Клиентам и серверам для работы с каналами допускается использовать функции из табл. 8.

Кроме того, для работы с каналами используется функция CreateFile (для подключения к каналу со стороны клиента) и функции WriteFile и ReadFile для записи и чтения данных в/из канала соответственно.

8. Функции работы с каналами

Функция	Описание
CallNamedPipe	Выполняет подключение к каналу, записывает в канал сообщение, считывает из канала сообщение и затем закрывает канал
ConnectNamedPipe	Позволяет серверу именованных каналов ожидать подключения одного или нескольких клиентских процессов к экземпляру именованного канала
CreateNamedPipe	Создает экземпляр именованного канала и возвращает дескриптор для последующих операций с каналом
CreatePipe	Создает безымянный канал
DisconnectNamedPipe	Отсоединяет серверную сторону экземпляра именованного канала от клиентского процесса
GetNamedPipeHandleState	Получает информацию о работе указанного именованного канала
GetNamedPipeInfo	Извлекает свойства указанного именованного канала
PeekNamedPipe	Копирует данные их именованного или безымянного канала в буфер без удаления их из канала.
SetNamedPipeHandleState	Устанавливает режим чтения и режим блокировки вызова функций (синхронный или асинхронный) для указанного именованного канала
TransactNamedPipe	Комбинирует операции записи сообщения в канал и чтения сообщения из канала в одну сетевую транзакцию
WaitNamedPipe	Ожидает, пока истечет время ожидания или пока экземпляр указанного именованного канала не будет доступен для подключения к нему

ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Реализовать с помощью механизмов межпроцессного взаимодействия (отображение файлов, почтовые ящики, каналы) одну из следующих задач в соответствии с вариантом:

Задача 1-3. Реализовать вычисление определителя квадратной матрицы с помощью разложения ее на определители меньшего порядка. При этом "ведущий" процесс рассылает задания "ведомым" процессам, последние выполняют вычисление определителей, а затем главный процесс вычисляет окончательный результат. Взаимодействие выполнить с помощью:

1 – отображения файлов;

2 – почтовых ящиков;

3 – каналов.

Задача 4-6. Реализовать нахождение обратной матрицы методом Гаусса, при этом задания по решению систем линейных уравнений распределяются поровну для каждого процесса. Взаимодействие выполнить с помощью:

4 – отображения файлов;

5 – почтовых ящиков;

6 – каналов.

Задача 7-9. Реализовать перемножение двух матриц с помощью нескольких процессов: каждый процесс выполняет перемножение строки первой матрицы на столбец второй (в соответствии с правилом умножения матриц). При необходимости процессам, выполняющим умножение, может быть отправлено несколько заданий. Взаимодействие выполнить с помощью:

7 – отображения файлов;

8 – почтовых ящиков;

9 – каналов.

Задача 10-12. Реализовать алгоритм блочной сортировки файла целых чисел. Каждый процесс, выполняющий сортировку, получает свою часть файла от ведущего процесса и сортирует его. Ведущий процесс выполняет упорядочивание уже отсортированных блоков. При необходимости ведомым процессам может быть выделено более одного задания на сортировку. Взаимодействие выполнить с помощью:

10 – отображения файлов;

11 – почтовых ящиков;

12 – каналов.

Задача 13-15. Реализовать обмен текстовыми сообщениями между несколькими процессами. Обеспечить возможность отправки сообщения сразу нескольким адресатам. Реализовать подтверждение приема сообщения адресатом или, в случае потери сообщения, повторную его передачу. Взаимодействие выполнить с помощью:

13 – отображения файлов;

14 – почтовых ящиков;

15 – каналов.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для решения каких задач может применяться межпроцессное взаимодействие?
2. Как можно осуществить межпроцессное взаимодействие, используя файловые отображения?
3. Как можно организовать параллельные вычисления при помощи именованных каналов?
4. В каких случаях необходимо использовать почтовые ящики?

СПИСОК ЛИТЕРАТУРЫ

1. Олифер, В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб. : Питер, 2002. – 544 с.
2. Грибанов, В.П. Операционные системы / В.П. Грибанов, С.В. Дробин, В.Д. Медведев. – М. : Финансы и статистика, 1990. – 239 с.
3. Соловьев, Г.Н. Операционные системы ЭВМ / Г.Н. Соловьев, В.Д. Никитин. – М. : Высшая школа, 1989. – 255 с.
4. Дейтел, Г. Введение в операционные системы. – В 2 т. / Г. Дейтел ; пер с англ. – М. : Мир, 1987. – Т. 1. – 359 с.
5. Фаронов, В.В. Delphi 2005. Язык, среда, разработка приложений / В.В. Фаронов. – СПб. : Питер, 2006. – 560 с.
6. Шеферд, Дж. Программирование на Microsoft Visual C++.NET / Дж. Шеферд, Д.Д. Круглински – СПб. : Питер, 2006. – 928 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ……... 3

Лабораторная работа 1. ФУНКЦИИ ПОЛУЧЕНИЯ СИСТЕМНОЙ ИНФОРМАЦИИ ……. 4

Лабораторная работа 2. АРХИТЕКТУРА WINDOWS ……. 12

Лабораторная работа 3. АРХИТЕКТУРА ПАМЯТИ WINDOWS …….. 25

Лабораторная работа 4. ПРОЦЕССЫ …… 37

Лабораторная работа 5. ПОТОКИ …….. 50

Лабораторная работа 6. МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ ... 63

СПИСОК ЛИТЕРАТУРЫ …… 74